Cray XD1™ FPGA Development

Private

S–6400–131

New Features

Cray XD1™ FPGA Development S–6400–131

This manual contains the following changes to reflect the changed support for FPGA development in Cray XD1 release 1.3:

- Updated development directory structure, including the new `makefile_vars` file
- Support in the API for the increased size of an Opteron memory region (up to 1 GB) that the FPGA can access
- New command, `xd1_jtag_route`(8), to establish a connection between any JTAG interface card port and any FPGA

In release 1.3.1, the manual was converted to the new format.

Record of Revision

Version	*Description*
1.3.1	October 2005 Supports Cray XD1 release 1.3.1 (1.3 general availability).
1.3	July 2005 Supports Cray XD1 release 1.3 (limited availability).
1.2	April 2005 Supports Cray XD1 releases 1.2 and 1.2.1.
1.1	October 2004 Supports Cray XD1 releases 1.1.
1.0	August 2004 Supports Cray XD1 releases 1.0.

Contents

Page

Page

Tables

Page

Figures

Preface

The information in this preface is common to Cray documentation provided with this software release.

Accessing Product Documentation

With each software release, Cray provides books and man pages, and in some cases, third-party documentation. These documents are provided in the following ways:

CrayDoc

The Cray documentation delivery system that allows you to quickly access and search Cray books, man pages, and in some cases, third-party documentation. Access this HTML and PDF documentation via CrayDoc at the following locations:

- The local network location defined by your system administrator
- The CrayDoc public website: `docs.cray.com`

Man pages

Access man pages by entering the `man` command followed by the name of the man page. For more information about man pages, see the `man`(1) man page by entering:

```
% man man
```

Third-party documentation

Access third-party documentation not provided through CrayDoc according to the information provided with the product.

Conventions

These conventions are used throughout Cray documentation:

Convention	Meaning
`command`	This fixed-space font denotes literal items, such as file names, pathnames, man page names, command names, and programming language elements.
variable	Italic typeface indicates an element that you will replace with a specific value. For instance, you may replace *filename* with the name `datafile` in your program. It also denotes a word or concept being defined.
`user input`	This bold, fixed-space font denotes literal items that the user enters in interactive sessions. Output is shown in nonbold, fixed-space font.
[ ]	Brackets enclose optional portions of a syntax representation for a command, library routine, system call, and so on.
. . .	Ellipses indicate that a preceding element can be repeated.
`name`(N)	Denotes man pages that provide system and programming reference information. Each man page is referred to by its name followed by a section number in parentheses. Enter: `% man man` to see the meaning of each section number for your particular system.

Reader Comments

Contact us with any comments that will help us to improve the accuracy and usability of this document. Be sure to include the title and number of the document with your comments. We value your comments and will respond to them promptly. Contact us in any of the following ways:

E-mail:
`docs@cray.com`

Telephone (inside U.S., Canada):
1–800–950–2729 (Cray Customer Support Center)

Telephone (outside U.S., Canada):
+1–715–726–4993 (Cray Customer Support Center)

Mail:
Software Publications
Cray Inc.
1340 Mendota Heights Road
Mendota Heights, MN 55120–1128
USA

Cray XD1 Support

Obtain support for the Cray XD1 product in either of the following ways:

Telephone:
1–888–279–2729 (Cray XD1 Customer Support Center)

Through the CRInform website:
`http://crinform.cray.com/xd/`

> **Note:** Use the contact information provided here if you have a support agreement with Cray. If, however, you have a support agreement with a third-party organization that is a Cray channel partner, contact that organization instead: do not contact Cray directly.

Introduction [1]

This chapter describes the intended audience for this manual and its scope, and lists the related publications.

1.1 Who Should Read this Manual

This manual is intended for an experienced hardware or field-programmable gate array (FPGA) designer who is proficient in logic design using HDL languages.

1.2 Scope of this Manual

This manual is a guide to the FPGA application acceleration processor, an optional component in a Cray XD1 system. It describes how this processor fits into the Cray XD1 hardware architecture, and provides guidelines on designing, developing, and integrating FPGA binaries for use in the system. It does not provide detailed procedures on using the application acceleration processors in an application program; for that information, see *Cray XD1 Programming* (S–2433).

This document has two companion documents that provide in-depth design details:

- *Design of Cray XD1 RapidArray Transport Core* (S–6411)
- *Design of Cray XD1 QDR II SRAM Core* (S–6412)

1.3 How this Manual is Organized

This manual consists of the following chapters and appendix:

- Chapter 2: Cray XD1 Architecture

  Describes the application acceleration processor's hardware environment and the surrounding Cray XD1 architecture.

- Chapter 3: Expansion Module

  Describes the interfaces on the expansion modules.

- Chapter 4: Quick Start

  Describes, at a high-level, how to load an FPGA binary onto the Cray XD1 system for integration with application software. It describes some of the command-line utilities and application programming interface (API) function calls that you can use during the software integration process. For more information on the API, see *Cray XD1 Programming* (S-2433).

- Chapter 5: Standard HDL Development Flow

  Provides an overview of standard development flows for HDL languages.

- Chapter 6: Design Considerations

  Discusses issues to consider when you design FPGA logic for the FPGA AAP, including a discussion of possible target applications, processor-FPGA communication, and FPGA memory resources.

- Chapter 7: Designing for the Cray XD1 System

  Describes specifics on designing for the Cray XD1 AAP. It includes details of the file and HDL code structure in the Cray Inc. design template, as well as methods that the C application can employ to interact and share data with user logic running in the AAP.

- Chapter 8: Simulation and Debugging

  Describes the simulation models and methods available to simulate and debug the AAP.

- Chapter 9: Troubleshooting

  Describes common issues and resolutions.

1.4 Related Publications

This section lists Cray and third-party publications that supplement the contents of this manual.

1.4.1 Cray XD1 Publications

Refer to the publications in Table 1, page 3 for related information about the Cray XD1 system.

Table 1. Related Cray XD1 publications

Publication title	Brief description
Cray XD1 Release Description (S–2453)	Identifies the main new features and enhancements in a particular release of the product. Includes information about the hardware, embedded software, and Linux-based software of the system.
Cray XD1 System Overview (S–2429)	Overview of the Cray XD1 computer and a description of its hardware and software components.
Cray XD1 Programming (S–2433)	Development tools on the Cray XD1 system and the application programming interface for the FPGA application acceleration processor.
Design of Cray XD1 RapidArray Transport Core (S–6411)	Companion to this document. Provides in-depth design details for a required IP core.
Design of Cray XD1 QDR II SRAM Core (S–6412)	Companion to this document. Provides in-depth design details for a required IP core.
Cray XD1 Release Notes (S–2455)	Information about resolved issues and known issues for a particular release of the product.

1.4.2 Third-party Publications

The Cray HDL template and reference designs are written in VHDL. Table 2, page 4 lists publications that are useful references for designers who work in VHDL.

Table 2. VHDL references

Title	Author, Publisher
VHDL for Programmable Logic	Kevin Skahill, Cypress Semiconductor
VHDL for Logic Synthesis	Andrew Rushton, John Wiley & Sons Ltd.
The Designer's Guide to VHDL	Peter J. Ashenden, Morgan Kaufmann Publishers

Cray XD1 Architecture [2]

This chapter describes the Cray XD1 system from a hardware perspective. It covers the high–level physical layout of a Cray XD1 system, as well as the RapidArray interconnect. This chapter focuses on the environment of the application acceleration processor, an optional component in a Cray XD1 system. The application acceleration processor is a field-programmable gate array (FPGA).

2.1 Cray XD1 High-level Physical Layout

The basic architectural unit of the Cray XD1 system is the Cray XD1 *chassis*. A large number of chassis may be connected together to form a Cray XD1 system.

2.1.1 Chassis

A Cray XD1 chassis may contain a maximum of 31 commodity and specialized processors that reside on a maximum of 6 compute blades. Figure 1, page 6 shows a physical view of the Cray XD1 chassis.

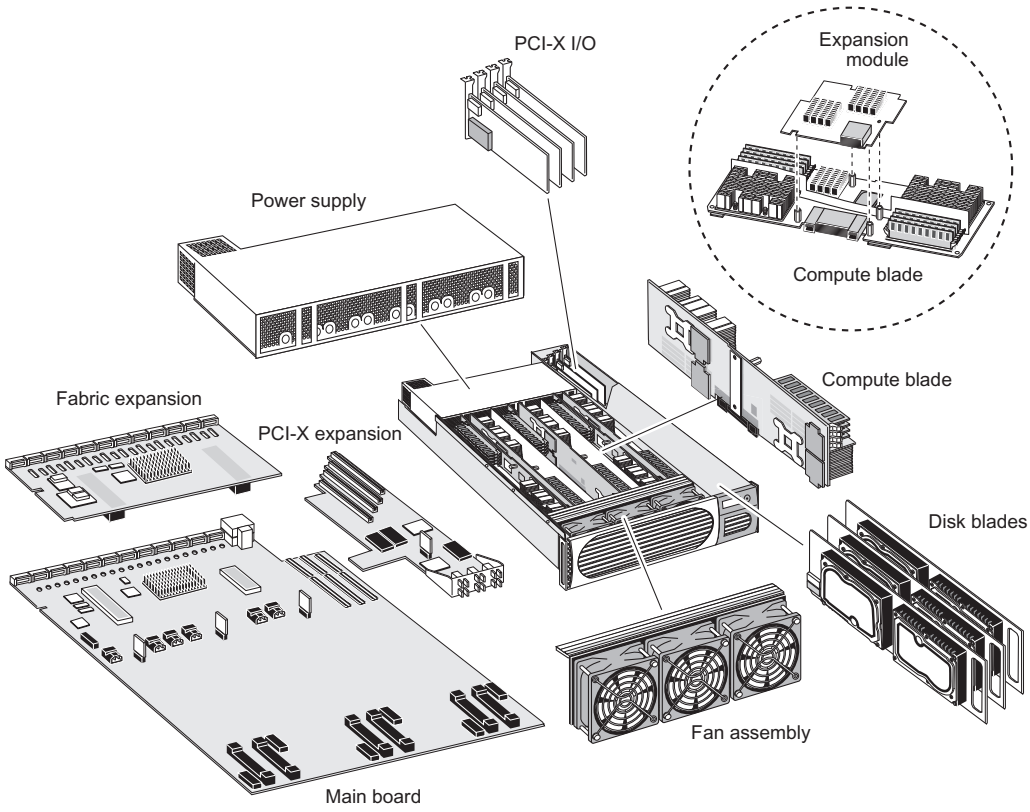

Figure 1. Cray XD1 chassis, physical view

In a fully populated chassis, these processors are distributed as follows:

- Twelve 64-bit AMD Opteron processors configured as six two-way symmetric multiprocessors (SMPs) that run Linux.
- Twelve RapidArray processors—Processors, designed by Cray, that process most of the communications within a chassis.
- Six application acceleration processors—Optional FPGAs that act as coprocessors to the Opteron processors.
- A management processor that runs software to monitor and manage the chassis hardware.

Figure 2, page 7 illustrates the logical distribution of these processors.

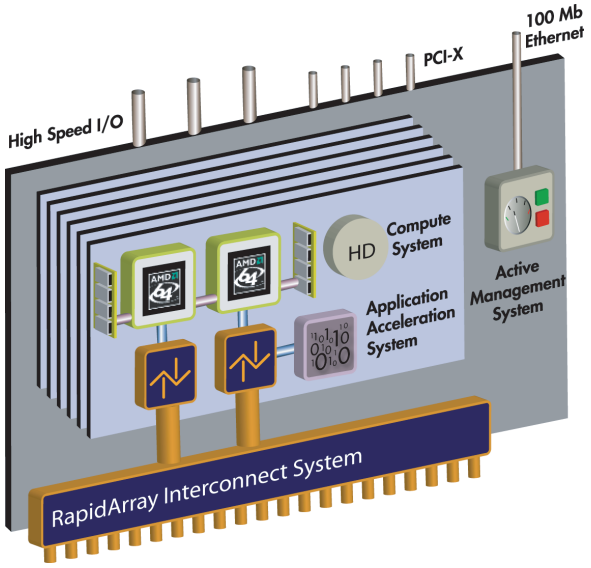

Figure 2. Cray XD1 chassis, logical view

A Cray XD1 computer *system* can consist of one to hundreds of Cray XD1 chassis that interconnect in various configurations. A maximum of 13 chassis reside in a seven-foot *cabinet*.

2.1.2 Compute Blade

A *compute blade* is a subassembly of the Cray XD1 chassis; refer to Figure 1, page 6 and to Figure 3, page 8. Each chassis contains one to six compute blades. Each compute blade includes the following components:

- Two 64-bit Opteron processors, configured as a two-way SMP
- 1 to 8 GB of double data rate synchronous dynamic random access memory (DDR SDRAM) per compute processor, providing a maximum of 16 GB of DDR SDRAM per SMP
- A RapidArray processor, which provides two 2-GBps RapidArray links to the switch fabric
- A connector for an expansion module (not shown in Figure 3, page 8)

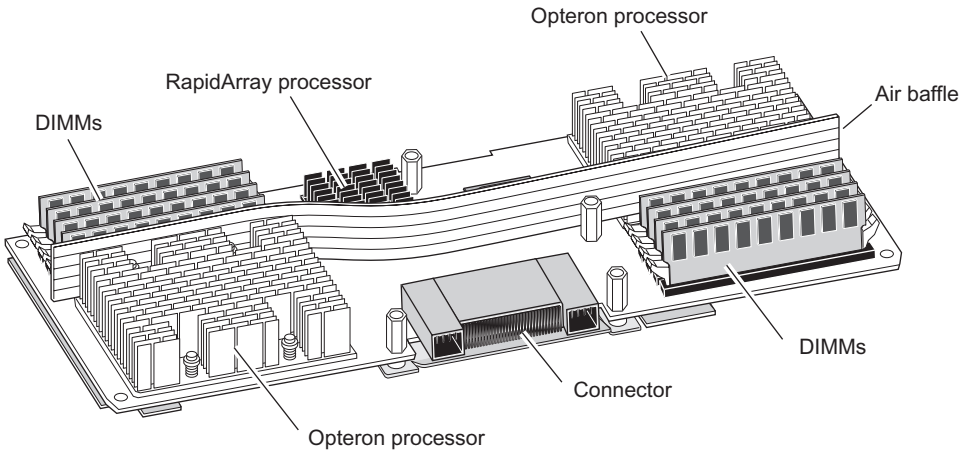

Figure 3. Compute blade

2.1.3 Expansion Module

The expansion module is an optional customer-orderable subassembly that attaches to a compute blade; refer to Figure 4, page 9 and Figure 5, page 9. Each expansion module contains the following components:

- An additional RapidArray processor, which provides two additional RapidArray links per compute blade
- An optional application acceleration processor (FPGA)
- Four QDR II SRAMs for the FPGA
- A programmable clock source for the FPGA

When all 6 expansion modules are fully populated, a chassis contains 12 RapidArray processors, 24 internal RapidArray links, and 6 application acceleration processors.

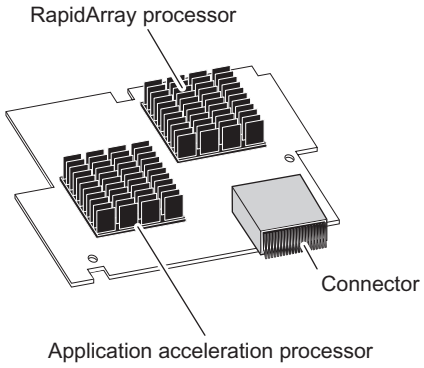

Figure 4. Expansion module, physical view

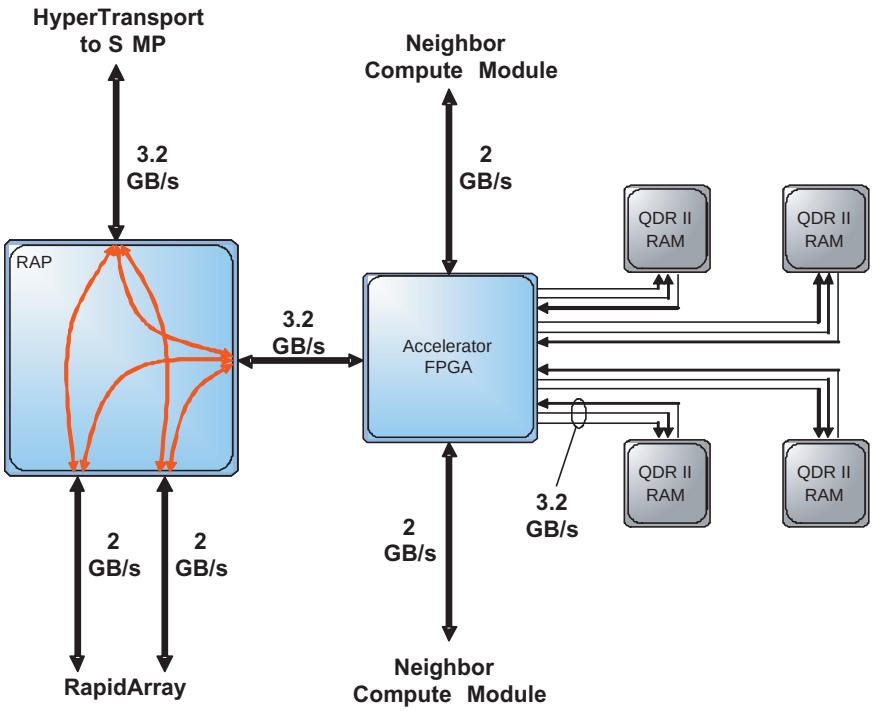

Figure 5. Expansion module, logical view

The RapidArray processor provides the interface for the FPGA to connect to the local Opteron processors as well as the RapidArray fabric. The interface's speed and performance match that of the HyperTransport links, but the link protocol is simplified to reduce logic requirements in the FPGA.

The quad-data-rate (QDR) static random access memory (SRAM) provides local high-speed storage for the FPGA. Each of the four QDR II SRAM circuits has its own fully independent control.

The programmable clock enables the user to set the speed of the FPGA for each design.

2.2 RapidArray Interconnect

Processors and memory within a chassis and between chassis are linked by a high-speed switch fabric called the RapidArray interconnect. Regardless of the size of a Cray XD1 system, the high-bandwidth, low-latency RapidArray interconnect is its central organizing construct. This interconnect enables the system to avoid PCI-X bus bottlenecks and shared-resource contention.

The RapidArray interconnect is a 96-GB-per-second (maximum per chassis) nonblocking, embedded crossbar-switch fabric that connects the RapidArray processors (RAPs). Each chassis has either one or two RapidArray switch fabrics, each of which consists of RapidArray links and a 24-port internal switch; see Figure 6, page 11.

The compute blades connect to the internal RapidArray switch at 4 GBps: 2 GBps each for simultaneous transmit and receive operations.

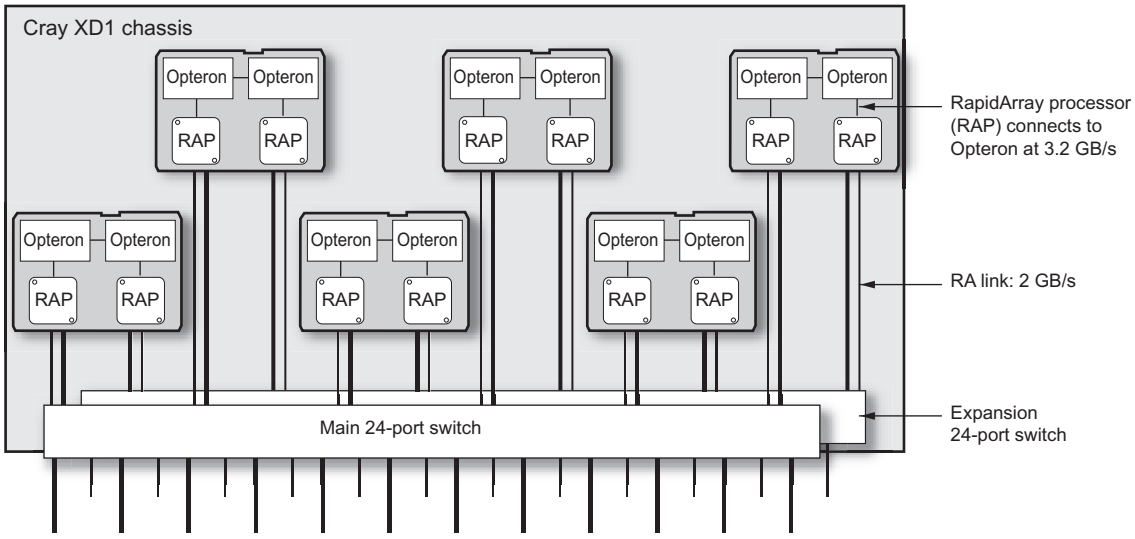

Figure 6. RapidArray components in a Cray XD1 chassis

To make use of the optional additional RapidArray components on the expansion modules, a chassis must also include a fabric expansion card; refer to Figure 1, page 6. The fabric expansion card provides the chassis with a second RapidArray switch. With this card and six expansion modules, a chassis has 24 external RapidArray interfaces; without it, only 12.

The optional expansion modules and fabric expansion card enable the compute blades to connect to the internal RapidArray switch at 8 GBps: 4 GBps each for simultaneous transmit and receive operations.

Expansion Module [3]

This chapter provides additional information about the expansion module—in particular, about the FPGA application acceleration processor (AAP). The manufacturer of the FPGA, Xilinx, Inc., also provides thousands of pages of documentation on its website at `http://www.xilinx.com`.

Table 3, page 13 lists the FPGA AAP modules that the Cray XD1 system currently supports.

Table 3. Current FPGA AAP variants

Cray part	Cray 87 no.	Xilinx model	Module memory
90-0003-05	87-0003-09	XC2VP50-7	4 x 1M x 36
90-0003-08[1]	87-0003-11	XC2VP50-7	4 x 1M x 36

Note: Cray 87 numbers are listed to allow translation between the Cray 87 number reported by some parts of the Cray XD1 software and the Cray part number that is used in most places. Use the Cray part number whenever possible because the software will not accept the Cray 87 numbers in future releases.

3.1 FPGA

The Xilinx Virtex-II Pro series of field-programmable gate array (FPGA) is arguably the most technically sophisticated silicon and software product in the programmable logic industry. Users can program the FPGA to execute computationally intensive and repetitive algorithms. Virtex-II Pro devices are optimized for high-density and high-performance system designs. The following subsections describe the types of functionality that these devices implement.

1 Educational version

3.1.1 Configurable Logic Blocks

Configurable logic blocks (CLBs) provide functional elements for combinatorial and synchronous logic, including basic storage elements. BUFTs (3-state buffers) associated with each CLB element drive dedicated, segmentable, horizontal-routing resources. Each CLB includes four slices (a maximum of 128 bits) and two 3-state buffers.

3.1.2 Block SelectRAM+ Memory Modules

Block SelectRAM+ memory modules provide large 18-Kb storage elements of dual-port RAM, programmable from 16K x 1 bit to 512 x 36 bits, in various depth and width configurations. Each port is totally synchronous and independent and offers three read–during–write modes. Block SelectRAM+ memory can cascade to implement large embedded-storage blocks.

The supported memory configurations for dual-port and single-port modes are: 16K x 1 bit, 8K x 2 bits, 4K x 4 bits, 2K x 9 bits, 1K x 18 bits, and 512 x 36 bits.

3.1.3 Embedded Multiplier Blocks

A multiplier block is associated with each SelectRAM+ memory block. The multiplier block is a dedicated 18 x 18 bit twos-complement signed multiplier. It is optimized for operations based on the block SelectRAM+ content on one port. The 18 x 18 multiplier can be used independently of the block SelectRAM+ resource. Read / multiply / accumulate operations and DSP filter structures are extremely efficient. Both the SelectRAM+ memory and the multiplier resource connect to four switch matrices to access the general routing resources.

3.1.4 Digital Clock Manager Blocks

Digital clock manager (DCM) blocks provide self-calibrating, fully digital solutions for clock distribution delay compensation, clock multiplication and division, and coarse- and fine-grained clock phase shifting. The DCM and global clock multiplexer buffers provide a complete solution for designing high-speed clock schemes.

A maximum of twelve DCM blocks are available. To generate de-skewed internal or external clocks, you can use each DCM to eliminate clock distribution delay. The DCM also provides 90-, 180-, and 270-degree phase-shifted versions of its output clocks. Fine-grained phase shifting offers high-resolution phase adjustments in increments of 1 / 256 of the clock period. Very flexible frequency

synthesis provides a clock output frequency equal to a fractional or integer multiple of the input clock frequency.

Virtex-II Pro devices have 16 global clock MUX buffers, with a maximum of eight clock nets per quadrant. Each clock MUX buffer can select one of the two clock inputs and switch glitch-free from one clock to the other. Each DCM can send a maximum of four of its clock outputs to global clock buffers on the same edge. Any global clock pin can drive any DCM on the same edge.

3.1.5 Virtex II Fabric (Routing and Interconnect)

A new generation of programmable routing resources called Active Interconnect Technology interconnect all elements. The general routing matrix (GRM) is an array of routing switches. Each programmable element is tied to a switch matrix, which enables multiple connections to the general routing matrix. The overall programmable interconnection is hierarchical and supports high-speed designs.

3.1.6 Embedded IBM PowerPC 405 RISC Processor Blocks

Each FPGA contains two embedded PowerPC processors. The PPC405 RISC CPU can execute instructions at a sustained rate of one instruction per cycle. On-chip instruction and data cache reduce design complexity and improve system throughput. Embedded PowerPC 405 RISC processor blocks provide maximum performance of 400 MHz.

All programmable elements, including the routing resources, are controlled by values stored in static memory cells. These values are loaded in the memory cells during configuration and can be reloaded an unlimited number of times to change the functions of the programmable elements.

3.2 QDR II SRAM Interface

The Cray QDR II IP core block controls data flow between the FPGA and the module's QDR II RAMs (refer to Figure 7, page 16). Each QDR II SRAM has its own QDR interface. On the SRAM (right) side of the circuit, the interface has separate read and a write buses that function independently and can simultaneously transfer 36 bits (32 bits of data and 4 bits of parity) at a rate of up to 1.6 GB per second in each direction. On the user (left) side of the circuit, the interface has separate read and write buses that function independently and can simultaneously transfer 72 bits (64 bits of data and 8 bits of parity) at a rate of one 64-bit word per clock period in each direction.

For design details, see *Design of Cray XD1 QDR II SRAM Core* (S–6412).

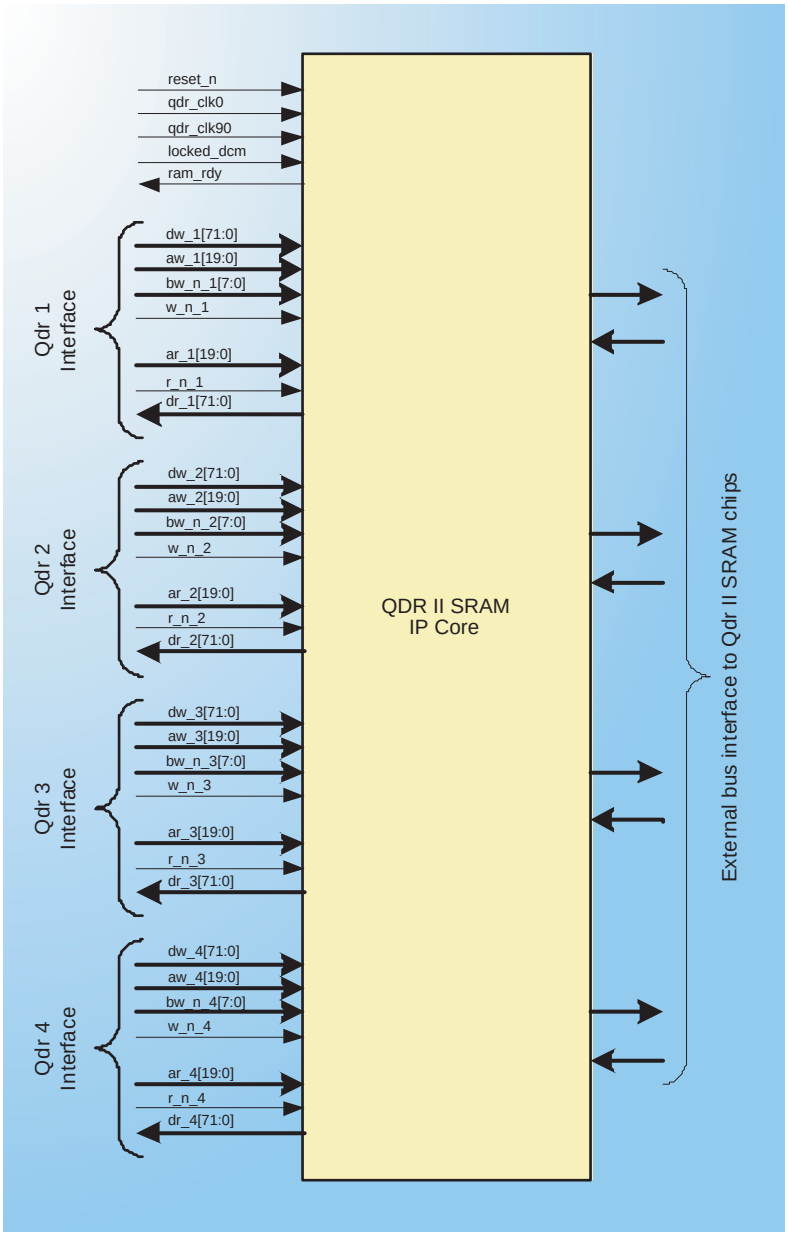

Figure 7. QDR II IP Core interface

3.3 RapidArray Transport Interface

The Cray RapidArray Transport (RT) IP core block provides the RapidArray fabric interface to an FPGA design; see Figure 8, page 18. The interface can both initiate and process responses for read and write transactions across the fabric. To facilitate this, there are two interfaces to the core: fabric request and user request.

The Fabric Request Interface issues read and write requests to the user logic when it receives packets from the fabric (which originate from a node). The User Request Interface accepts read and write requests from the user logic that are for a device on the fabric (SMP processor memory).

The RT interface has the following characteristics:

- 64-bit interface at a maximum speed of 200 MHz
- 3.2 GBps interface—1.6 GBps simultaneous transmit and receive
- Posted writes
- Multiple outstanding read requests
- Data bursts up to 64 bytes per request

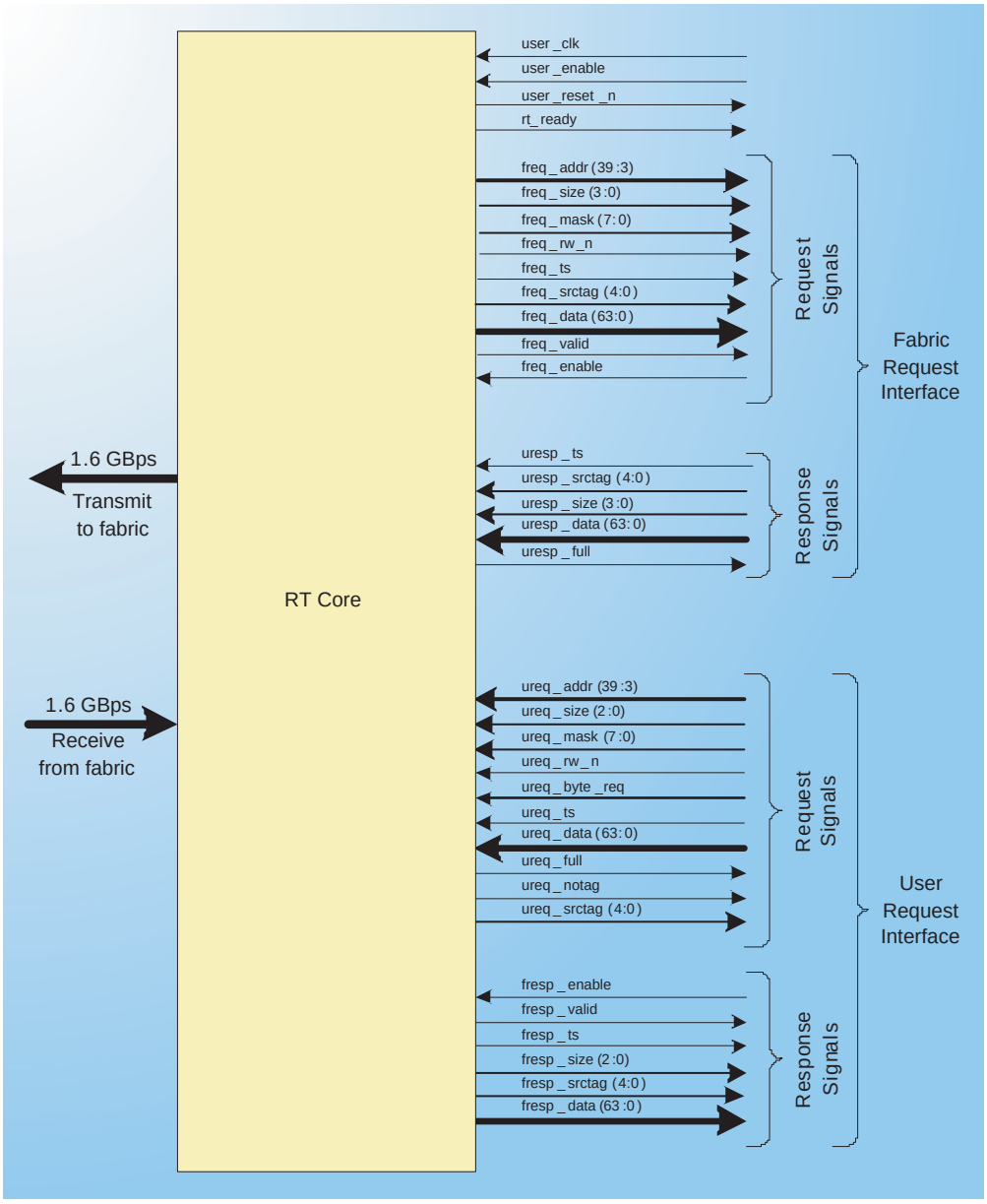

Figure 8. RT interface

Quick Start [4]

This chapter describes the Cray FPGA reference designs and how to use them along with the `fcu`(1) utility program to explore the FPGA subsystem. Many of the techniques described are useful during initial integration between a completed FPGA design and the software that uses it.

4.1 Reference Design Overview

The following reference designs are in subdirectories of `/opt/ufpapps` if the `ufpapps` software package is installed (this is the case if the node is in a partition with the "full" software installation option):

- Hello

  A simple design that demonstrates the basic operation of interfacing the FPGA to the Opteron processor. This design is an excellent place to start for both software and FPGA designers.

- Mince

  The Minimal Compute Engine (Mince) design demonstrates interactions between the Opteron processor and FPGA that are more complicated. Mince is a diagnostic tool that tests the FPGA memory and bus interfaces.

- MTA

  The Mersenne Twister Accelerator (MTA) design demonstrates a practical application, an FPGA implementation of the Mersenne Twister pseudorandom number generator.

Documentation for each reference design and its sample application programs is in the `doc` subdirectory (for example, `/opt/ufpapps/mince/doc`) of the relevant reference design directory.

4.2 Command-line Manipulation of FPGAs

This section describes the process that you use to manipulate the FPGA from a Linux shell.

This tutorial focuses on the Mince design because it is a good diagnostic tool. You can use it to verify that the FPGA subsystem is working correctly. You can use a similar procedure with any of the reference designs.

4.2.1 Overview

After you simulate an FPGA application design and compile it into binary form, you can load it onto the FPGA AAP. The Cray XD1 system provides a variety of Linux command–line tools and a software API to facilitate this process. A detailed description of the FPGA software API, and command-line utilities is available in *Cray XD1 Programming* (S–2433).

4.2.2 Copying the Design Directory

The reference designs are in `/opt/ufpapps`. This procedure modifies some of these files. Therefore, you must first copy them to a user-writable location such as `$HOME` or `/tmp`.

1. Log in to Linux on the Cray XD1 system.
2. Copy the design directory structure:

   > **`cp -rP /opt/ufpapps $HOME`**

3. Make the design user-writable:

   > **`chmod -R u+w $HOME/ufpapps`**

4. Go to the `mince` design directory:

   > **`cd $HOME/ufpapps/mince`**

5. Go to the location of the compiled binary files:

   > **`cd bin`**

4.2.3 Converting FPGA Binary Files

The final output from the FPGA design process is an FPGA binary file that contains the configuration bit stream for a specific Xilinx device. Before you can load an FPGA binary on the FPGA AAP, you must convert it to a Cray proprietary format by prepending a header. The header contains information about both the hardware for which the design is targeted and the clock rate at which to run the design. You can perform this task with the FPGA control utility (the `fcu`(1) command). It is a general purpose command that enables a designer to program and interact with the FPGA. For a list of options, refer to the `fcu`(1) man page or type **`fcu --help`**.

To convert the FPGA binary, the `fcu`(1) command prepends information from a text file to the Xilinx binary file *`design`*`.bin` and creates the Cray proprietary file *`design`*`.bin.ufp`. If no header file exists, the `fcu` command can create it.

The following example creates a new header file and converts `mince50.bin` to `mince50.bin.ufp`. This example assumes that the FPGA AAP is an XC2VP50 Xilinx chip.

1. Log in to Linux on the Cray XD1 system.
2. Determine the type of application accelerator on the target node:

```
> lsnode -v | grep -E -i '^hardware|^app'
Hardware ID:                  269.1
App Accelerator:              87-0003-09
Hardware ID:                  269.2
App Accelerator:              87-0003-09
Hardware ID:                  269.3
App Accelerator:              87-0003-09
Hardware ID:                  269.4
App Accelerator:              87-0003-09
Hardware ID:                  269.5
App Accelerator:              87-0003-09
Hardware ID:                  269.6
App Accelerator:              87-0003-09
```

3. If necessary, use Table 3, page 13 to determine the part number. For example, 87-0003-09 matches part number 90-0003-05.

4. Create a header file for the target application accelerator:

```
> fcu -b
Enter Cray Part Number [10 characters]
90-0003-05
Enter FPGA clock frequency in Integer MHz
 [Range : 63 to 199, inclusive]
180
```

   The `fcu` command creates a text file called `ufphdr`. The contents of the file are as follows:

```
Cray Part Number   : 90-0003-05;
FPGA Frequency MHz : 180;
```

5. Convert the Xilinx binary file to the Cray format:

```
> fcu -c mince50.bin ufphdr
header size 59
input fpga load size 2377668
```

   In this example, the `fcu` command creates a file called `mince50.bin.ufp` which is ready for download.

4.2.4 Downloading Files to the FPGA

You can load an FPGA *design*`.bin.ufp` file through either the software API function `fpga_load`(3) or the `fcu`(1) utility from the command line. To load a file onto the FPGA from the command line, use the `fcu` command as in the following example:

```
> fcu -l mince50.bin.ufp
file size 2381764
setting device location /dev/ufp0
programming device 2381764 bytes
opening device /dev/ufp0
programmed FPGA 2381764 bytes
closing device file descriptor 4
```

The FPGA is now programmed with the `mince50.bin.ufp` logic.

4.2.5 Accessing the FPGA

In general, access to the FPGA occurs through the Linux API. The API provides functions for reading and writing values to the FPGA, and for mapping the QDR SRAM of the FPGA into the virtual memory space of the processor.

As a convenience, a virtual file `/proc/ufp/regs` exists to show the contents of some configuration registers that are not directly user-accessible and the first 64 bytes of the QDR SRAM register space. This can be useful for displaying design-dependent configuration and status registers from the command line during initial debugging. Bit definitions for the Host Interface registers are in *Design of Cray XD1 RapidArray Transport Core* (S–6411). An example of the virtual file contents follows:

```
> cat /proc/ufp/regs
LPC Bus Kernel Virtual Address ffffff0040df8000

FPGA AAP Config and Status Space
Offset 0x200
CME Ctrl Reg         : 0x05
CME Stat Reg         : 0x03
CME Intr Status Reg  : 0x07
Host Interface
Offset 0x300
RAT Base Num Reg 0   : 0x01
RAT Base Num Reg 1   : 0x00
RAT Config Num Reg   : 0x01
RAT Version Num Reg  : 0x00
Test Reg 0           : 0x00
Test Reg 1           : 0x00
RAT Compile Num Reg  : 0x04
Clock Config Reg 0   : 0xB4
User Reset           : 0x01
Clock Status Reg 0   : 0x00
Clock Frequency MHz  : 0xB3
Host Latch Register  : 0x00

FPGA AAP Application Space Starting Kernel Virtual Address
ffffff0040dfa000

Application Interface
Register space kernel virtual address ffffff0044dfa000
Offset Range  : Value
0x0000-0x0008 : 0x0000002B00080100
```

```
0x0008-0x0010 : 0x0000000000000000
0x0010-0x0018 : 0x0000000000000000
0x0018-0x0020 : 0x0000000000000000
0x0020-0x0028 : 0x0000000000000000
0x0028-0x0030 : 0x0000000000000000
0x0030-0x0038 : 0x0000000000000000
0x0038-0x0040 : 0x0000000F0000000E
```

The first two tables of registers are of little general interest, except the `Clock Config Reg 0` and `Clock Frequency MHz` registers of the Host Interface. The clock configuration register shows the requested programmable clock frequency (a hexadecimal value in MHz) at which the FPGA application will run. The clock frequency MHz register shows the measured clock frequency with an accuracy of +/– 1 MHz.

The Application Interface registers are the first 64 bytes of the FPGA register space. Their content depends on the design of the application logic. These values are refreshed every time you read the file.

4.2.6 Erasing the FPGA

After you program an FPGA, it remains operational until you reset or unload it. Resetting the FPGA leaves the device configured but holds the user logic in reset. If you take the FPGA out of reset, it resumes normal operation. Unloading the FPGA clears all configuration in the device and returns it to an unprogrammed state.

Note: If you plan to continue to the next section, do not reset or unload the FPGA at this time.

To reset the FPGA:

```
> fcu --reset
```

To unload the FPGA:

```
> fcu --unload
```

4.3 Running the C Reference Programs

After you load an FPGA, it is ready for software to use it. Each reference design includes one or more programs that use the FPGA logic. Compiled versions of these programs are located with the design binary files in the `bin` directory of the

design. The Mince design includes `bertest`, `qdrtest`, and the `test.sh` script which calls `berttest` and `qdrtest`.

The following examples show calling the `berttest` program with two slightly different sets of parameter values. Both calls use the Mince FPGA to run a bit error rate test (BERT) on the QDR SRAM and on a segment of the SMP DRAM (to test the communication channel between the FPGA and the SMP memory).

```
> ./berttest -v -r -w 0 -t 60 -d rand -a incr
Test Parameters -
  Data Pattern     : rand
  Address Pattern  : incr
  Wait States      : 0
  QDR RAM Banks    : Bank0 Bank1 Bank2 Bank3
  RT BERT          : Enabled
  Execution Time   : 60 s
Test PASSED.
> ./berttest  -v -r -w 0 -t 60 -d incr -a rand
Test Parameters -
  Data Pattern     : incr
  Address Pattern  : rand
  Wait States      : 0
  QDR RAM Banks    : Bank0 Bank1 Bank2 Bank3
  RT BERT          : Enabled
  Execution Time   : 60 s
Test PASSED.
```

The shell script `test.sh` calls both `berttest` and `qdrtest` numerous time with different parameters. This script is a useful tool for performing a sanity check on the AAP hardware.

Standard HDL Development Flow [5]

This chapter describes a standard development flow that begins with design entry and ends with an FPGA binary.

5.1 Overview of the Development Process

Cray Inc. uses standard processes and tools to develop designs for the FPGA application acceleration processor. The overall development process has the following four basic steps:

1. Design entry—Create the source code that makes up the design.
2. Synthesis—Translate the source code into a low-level netlist of logic elements that are suitable for implementation in an FPGA.
3. Simulation—Simulate the design source code or the synthesized design netlist.
4. Implementation—Determine the physical placement and routing for the netlist logic on the target FPGA.

A designer has a wide variety of Xilinx and third-party tools to choose from for FPGA development (Cray does not supply them). The following sections identify some of the most common options for each of the development steps. The following figure illustrates the standard development flow.

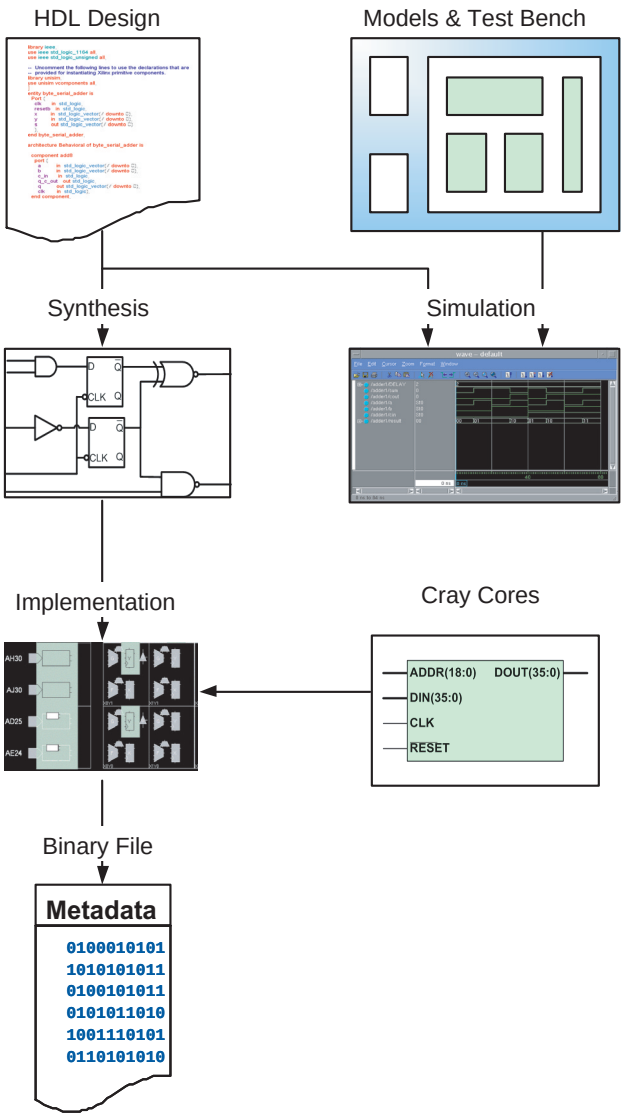

Figure 9. Standard development flow

5.2 Design Entry

Hardware description languages (HDLs) can specify a hardware design in terms of familiar programming constructs such as conditional statements, loops, and function calls. VHDL and Verilog are perhaps the two most common languages in current use. Because they are tailored specifically to hardware design, HDLs provide a flexible and powerful way to generate efficient logic. However, this tailoring makes them unfamiliar territory for people outside the hardware design field.

In order to communicate hardware design to a more general audience, a number of companies are working to support the use of other programming languages (primarily C and C++) as HDLs. The development of C and C++ for hardware design facilitates the partitioning of resources between software and hardware, and facilitates hardware and software co-simulation and code reuse.

You can generate the user application block with many tools, including higher-level languages such as C and C++ or the Xilinx System Generator for DSP package. The latter enables you to design digital signal processing (DSP) blocks with the MATLAB package from The MathWorks. Figure 10, page 30 shows how a higher-level flow might fit into the standard development flow of the Cray XD1 system.

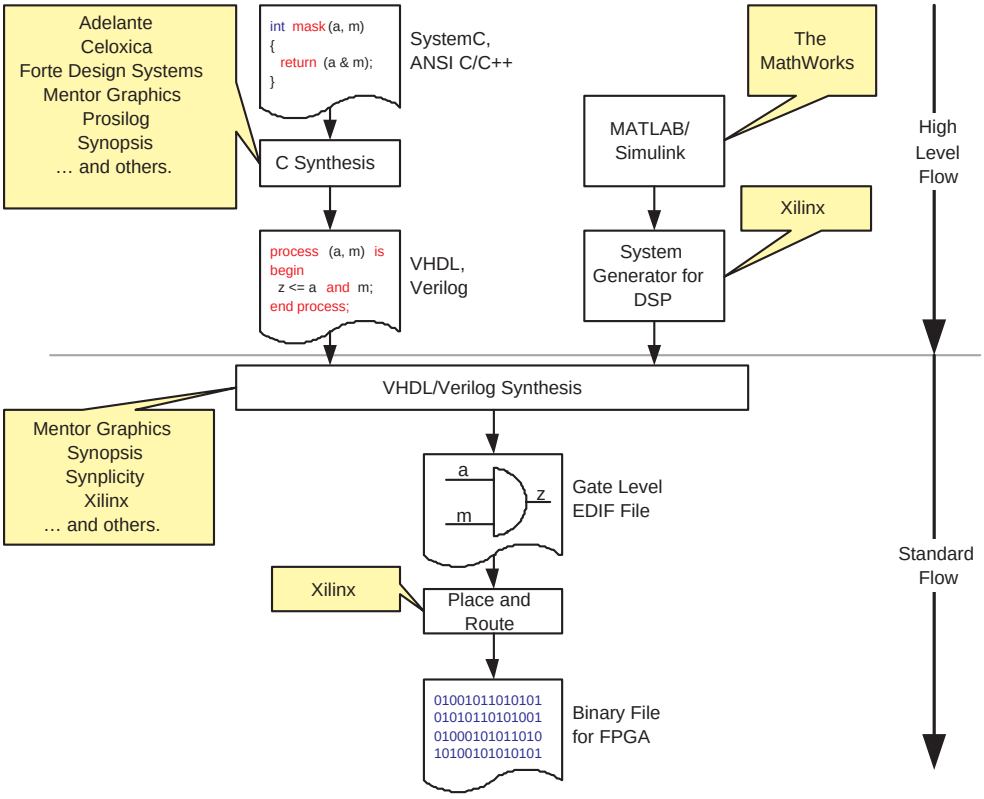

Figure 10. Additional high-level tools

The Xilinx ISE tools provide a text editor for generating design source code. The text editor includes support for highlighting both VHDL and Verilog syntax. It also includes templates for common logic functions. Several other free text editors support design entry; for example, GNU Emacs (or Xemacs). Like the Xilinx ISE software, Emacs supports both a VHDL mode and Verilog mode that provide syntax highlighting and design templates.

In addition to designing and coding logic by hand, you can use the Xilinx CORE Generator tool in ISE to create custom logic blocks from a family of parameterizable cores. A wide variety of common cores is available, including simple mathematical functions and DSP functions.

5.3 Synthesis

The synthesis process translates the source HDL code into gate-level elements, such as AND gates, OR gates, and flip-flops. The Cray reference designs use the Xilinx XST synthesizer (part of the Xilinx ISE package). However, synthesis tools are also available from several other vendors; for example: Synplicity, Mentor Graphics, and Synopsis.

The output of the synthesis process is a gate-level netlist. If you use the Xilinx XST package, the netlist is a Xilinx proprietary NGC file (for example, `top.ngc`).

5.4 Simulation

You perform the simulation step on a design test bench. A test bench is an HDL structure that includes the design itself along with models that manipulate the design inputs and monitor the outputs. Simulators compile HDL code into simulation models, which you can use to verify the design. If an FPGA design is fully implemented, you can create a back-annotated simulation model. This annotated model contains the actual delays from the placed design.

The reference designs work with either the Aldec Riviera simulator or the Mentor Graphics Modelsim simulator. However, most standard simulators should work because the designs, test bench, and models are all VHDL source files. You can also use more powerful test bench and simulation tools such as the Versity Specman simulator.

5.5 Implementation

Design implementation is the process of translating, mapping, placing, routing, and generating an FPGA binary file for your design. The implementation process uses the proprietary Xilinx translate, map, and place-and-route tools, which are part of the Xilinx ISE tool set, to assign the logic that you create during design entry and synthesis to specific physical resources of the target device.

A set of user-determined timing and placement constraints guides the Xilinx process. These constraints are in the user constraint file (UCF).

Design Considerations [6]

6.1 Advantages and Disadvantages of FPGAs

The power of a field-programmable gate array (FPGA) as an application acceleration processor is its ability to perform well in areas that a microprocessor does not, rather than its ability to compete directly with a standard microprocessor. The two main advantages an FPGA has over a microprocessor are as follows:

- FPGAs have a flexible architecture. You can customize and optimize the logic in the FPGA to perform only the required tasks.
- FPGAs can exploit parallelism. General-purpose microprocessors have limited support for instruction or data parallelism. At most, they can perform a handful of operations in parallel and this is difficult to control from a high level programming language. FPGAs can perform thousands of operations in parallel.[1]

Figure 11, page 34 illustrates that the 64-bit compute processor is limited to linear execution and fixed word length; the FPGA can manipulate multiple variable length data items concurrently by using fine-grained parallelism.

[1] This obviously depends upon the size and complexity of the operation, the size of the FPGA, and the opportunities for parallelism in the algorithm.

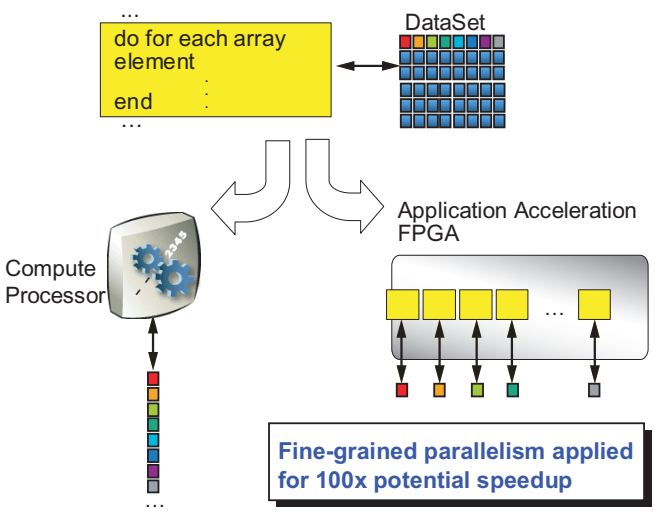

Figure 11. FPGA executes multiple codes simultaneously

An FPGA also has disadvantages. The key disadvantage is speed. A microprocessor has a fixed set of highly optimized functional blocks that can run at a very high clock rate. The generic logic building blocks and interconnect of an FPGA cannot support the same clock speeds. As a general rule, an FPGA supports a maximum clock rate that is approximately one–tenth that of a processor like the Opteron.

6.2 Target Applications

This section specifies the characteristics of target applications in general and describes one sample application.

6.2.1 Characteristics of Target Applications

The following characteristics of applications make them candidates for acceleration by FPGAs:

- The processor cannot reach its maximum performance due to loop overhead, cache thrashing, inefficient instructions, and so on.
- The processor cannot exploit data or instruction parallelism that is inherent in the function.

Some examples of real-world applications that exhibit these characteristics are as follows:

- Searching (generic data streams as well as specialized searches for genetic application)
- Sorting
- Digital signal processing
- Image processing and recognition
- Graphics acceleration
- Encryption and decryption
- Coding and decoding
- Error correction
- Random number generation
- Some mathematical algorithms
- Bit manipulation

6.2.2 Sample Application

The Mersenne Twister random number algorithm can efficiently generate a large number of pseudorandom numbers for use in computations such as Monte Carlo analysis. The FPGA uses this algorithm to generate integers with a uniform distribution that do not repeat a value in 2^{19937} -1 operations.

The FPGA delivers slightly more than three times the performance of the fastest available Opteron processor (at the time of writing). In addition, while the Opteron processor uses its full capacity to execute the algorithm, the FPGA uses less than 20% of its capacity.

The performance of the FPGA is limited only by the speed at which numbers can be written into processor memory, not by the FPGA logic. Figure 12, page 36 illustrates the path that the FPGA uses to access the compute processor memory.

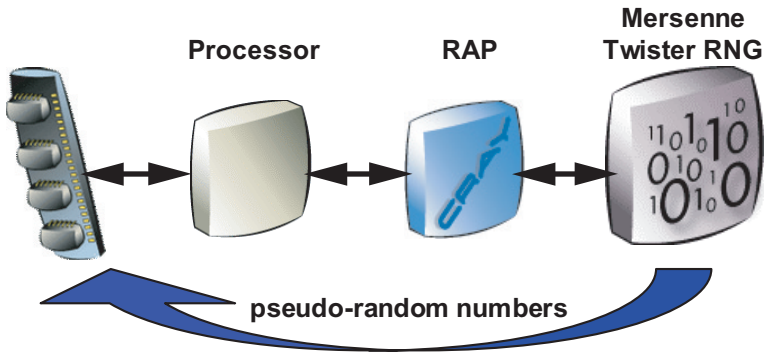

Figure 12. Random number example

Because the design consumes only a small amount of the FPGA, a great deal of capacity remains to provide post-processing functions for the random numbers. For example, the logic can convert the integer values to floating point values, or it can convert the uniform distribution to a normal distribution in an additional pipeline stage. An Opteron processor must perform these functions serially, which further degrades its performance.

Table 4, page 36 compares the random-number-generation capabilities of an Opteron processor and an FPGA.

Table 4. Results of random number generation

	2.2 GHz Opteron Processor	**FPGA (XC2VP50-7) @ 200 MHz)**
Source	Original C code	VHDL code
Speed (32-bit integers per second)	~101 million	~319 million
Capacity used	100%	< 20% (includes RT core)

6.3 FPGA Memory Resources

For many applications, access to memory resources can limit performance. The FPGA has access to four types of memory; this helps to maximize the performance of memory-intensive applications. The first two types of memory are the distributed and block RAM available within the Xilinx FPGA. These

small, high-speed memories are excellent for exploiting parallelism in algorithms because many of them are distributed across the chip.

The third type of memory is the four external banks of the second generation Quad Data Rate (QDR II) SRAM (refer to Figure 13, page 37). These memories provide high data transfer rates with low latency and no restrictions on burst size. In addition, they have independent read and write interfaces, so data can be simultaneously written and read from different addresses every clock cycle.

The last type of memory is the DDR SDRAM banks of the SMPs. The FPGA has high-speed burst access through the RapidArray fabric to the large pools of memory that belong to each compute processor. Figure 13, page 37 shows the memory hierarchy (slowest to fastest speed) that is available to the FPGA and the relative size of each type of memory.

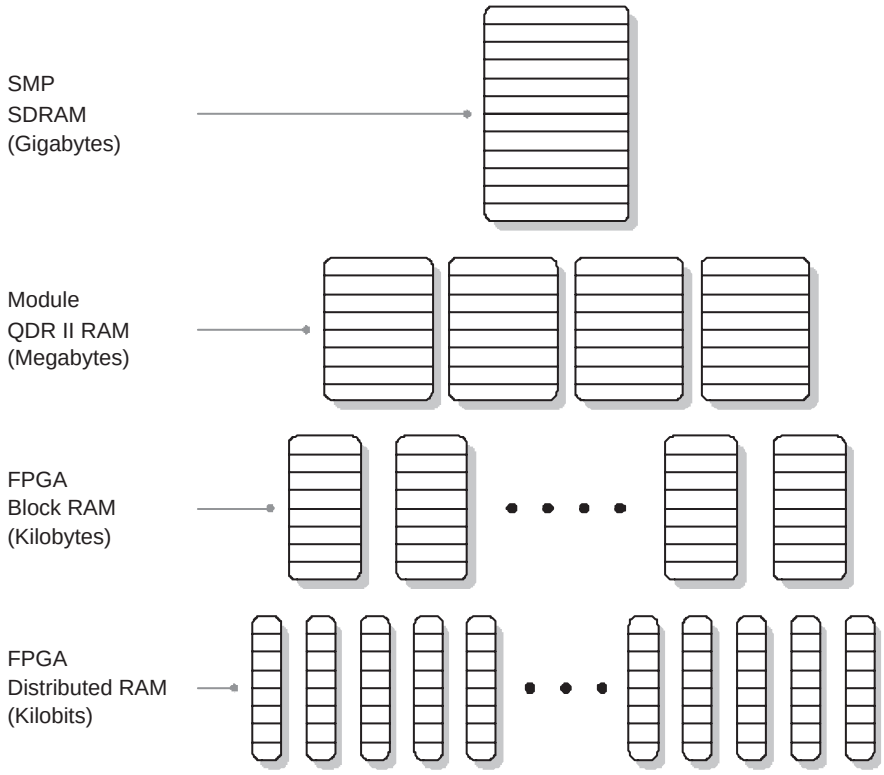

Figure 13. FPGA memory hierarchy

The types of memory have different characteristics that make them suitable for different tasks. Table 5, page 38 lists those characteristics and suggests some typical uses.

Table 5. Available memory devices

RAM Type	Characteristics	Typical uses
Opteron DDR SDRAM	Large pool of memory capable of high-speed burst access.	• Buffer large sets of data for processing as well as storage of the results
QDR II RAM	Medium-size pool of memory capable of high- speed random access. Supports simultaneous read and write accesses. Four banks allow independent parallel access.	• Store segments of a larger data set that is in Opteron SDRAM • Large lookup tables
FPGA Block RAM	Very-high-speed full dual- port RAM. Between 136 and 232, 18-Kb RAMs per chip. Enables a great deal of parallelism.	• FIFOs • Shift registers • Lookup tables
FPGA Distributed RAM	Very-high-speed pseudo dual-port RAM. 30,000 to 50,000 16x1 RAMs. Enables massive parallelism when memory requirements are small.	• Small FIFOs • Shift registers • Small lookup tables

6.4 Fabric Bandwidth and Data Flow

The RapidArray Transport bandwidth available to the FPGA depends on a number of factors. The theoretical maximum available bandwidth is approximately 8/9*1.6 Gigabytes per second (approximately 1.422 GBps) in each direction. This maximum bandwidth occurs only with efficient burst transactions between the Opteron and FPGA. The maximum write data bandwidth is decreased when the read interface is in operation, since read requests must be transmitted on the write bus. Another important consideration is the expansion fabric. The expansion fabric and the FPGA share a common path to the Opterons and so FPGA bandwidth can be affected by Opteron traffic that runs through the expansion fabric.

While the data path between the Opteron and FPGA is important, it is not the only path to consider. Transferring data into the Opteron memory from disk or a network location is much more likely to be the limiting factor in overall application performance.

6.5 SMP Processor-initiated Fabric Transactions

Writes made from the SMP processor to the FPGA are posted. This means that once the write is made, the processor moves on to its next operation. It does not wait for a response from the FPGA. This greatly improves the overall performance of transferring data to the AAP. Read requests, however, require the processor to wait for a response from the FPGA (in other words, the requested data). There is no mechanism for the processor to issue burst read requests to the FPGA or to have multiple outstanding read requests. As a result, the SMP processor can write to the FPGA much more efficiently than it can read. This is generally true for most types of processor interfaces, even, for example, DRAM interfaces. It is this performance gap between read and write transactions that gives rise to the popularity of write-only architectures for achieving the highest performance. In a write-only architecture, data is written from the SMP processor to the FPGA, and then the AAP writes it back to the SMP processor (rather than having the SMP processor read it back).

6.6 FPGA-initiated Fabric Transactions

The FPGA has complete control of its fabric interface and so it has capabilities not available to the application that runs on the SMP processor. The FPGA can burst read and burst write to and from the fabric (for example, to and from the SMP memory). In addition to the ability to post writes, the FPGA can also have multiple read requests outstanding. These facts can make DMA access from the FPGA very efficient.

6.7 Limitations

The current software release has an important restriction. The Opteron can burst write to the FPGA memory space, but cannot burst read from it. This issue can be addressed by programming the FPGA to write to the Opteron memory window, since the FPGA is capable of bursting in both directions.

You must run the QDR II SRAM IP core at a minimum clock speed of 130MHz. This restriction results from the output timing of the QDR II SRAMs. When the SRAMs operate at clock speeds of 130 MHz or above, an internal delay-locked loop (DLL) is enabled. When the SRAMs operate below 130 MHz the internal DLL is not used and the output timing of the data changes. The QDR II SRAM IP core does not adapt to the change in data output timing at clock speeds below 130 MHz. If the user logic must run at a clock speed slower than 130 MHz, the user logic must translate between the two clock domains.

Designing for the Cray XD1 System [7]

This chapter describes the Cray XD1 FPGA application acceleration processor development environment and how the FPGA user logic interfaces to the Cray XD1 system and to the application software.

7.1 Overview

User logic for the FPGA AAP must be able to communicate with the surrounding system. Cray IP cores provide this ability. Each core provides an interface to a specific part of the system. You must design the FPGA user logic to interact with these cores. The documents listed in Section 1.4.1, page 2 provide full descriptions of the Cray IP cores and their interfaces.

Cray includes a package (`ufpapps`) in the Cray XD1 software distribution to assist you in understanding how to integrate user logic into the Cray XD1 system. This package includes the Cray HDL cores, reference designs, and a design template. By default, the `ufpapps` RPM package creates a directory structure under `/opt/ufpapps`; see Figure 14, page 42.

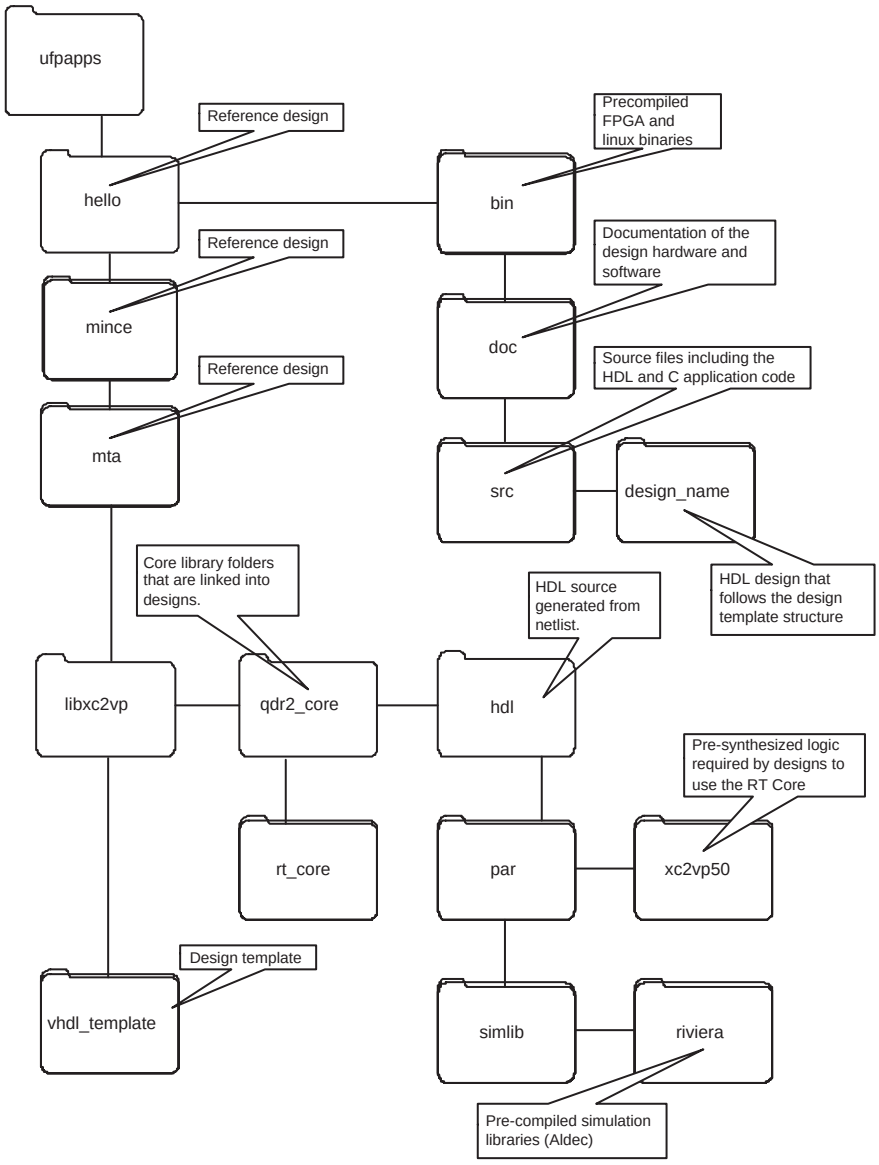

Figure 14. Structure of `/opt/ufpapps`

Each reference design is based on the design template and includes the Cray cores by reference in order to interface with the rest of the system. The hello design is a simple example and a good place to start becoming familiar with the HDL structures and the Cray cores. Table 6, page 43 describes the `ufpapps` directory structure.

Table 6. `ufpapps` directory descriptions

Directory	Subdirectory	Description
`hello`	`bin`	Contains precompiled FPGA binaries and C application code that can be used to quickly start interacting with the AAP.
	`doc`	Contains detailed documentation on the reference design HDL and C application code.
	`src`	Contains the C application source code and Makefile as well as a directory structure containing the HDL sources.
`mince`		Another reference design.
`mta`		Another reference design.
`libxc2vp`	*ipcore*`/par/xc2vp50`	Contains presynthesized logic that designs that use *ipcore* require. Cores are in NGC format because it can contain logic placement information. This helps to ensure consistent performance when you incorporate the cores in user designs.
	ipcore`/hdl`	Contains the synthesized netlists for the QDR II core. You can use this netlist to simulate the core, but it is slower than the simulation libraries.
	ipcore`/simlib/riviera`	Contains precompiled simulation libraries for Riviera.
`vhdl_template`		Contains the HDL design template described later in this section.

7.2 Design Template Structure

The design template in the `ufpapps` package includes a top-level VHDL file that instantiates all the components shown in `top.vhd` of Figure 15, page 44. The top-level design files of the template are in VHDL. However, by using VHDL's ability to support "black box" logic components, it is also possible to use other HDLs to develop designs for the AAP.

The top-level VHDL file forms a wrapper that combines several logic components. Two of the components are the required logic cores for connecting to the RapidArray processor and to the QDR II SRAMs. A third component generates the user-programmable clock. The fourth component is the user application component that contains all design-specific logic. The user application component can be written in VHDL, or Verilog, or any other language that can generate a Xilinx netlist. The only requirement is that the user logic component provides the required interface signals to the rest of the design.

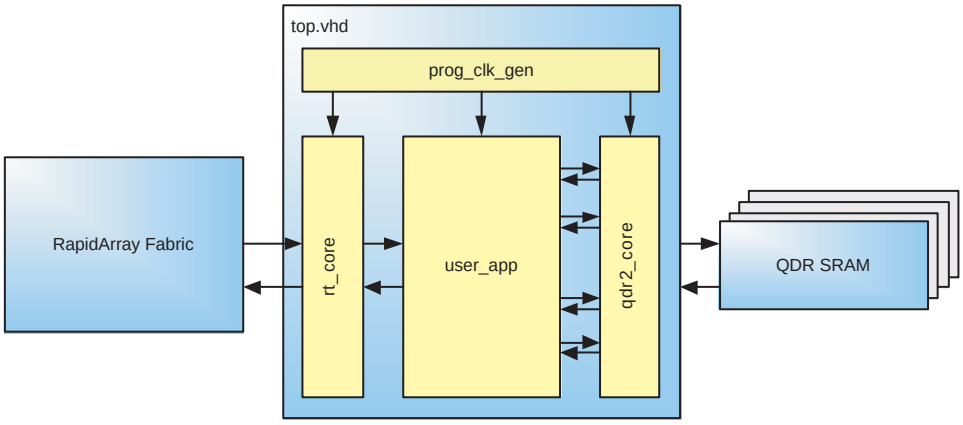

Figure 15. FPGA organization

The design template also provides a common directory structure and build environment that supports development with the graphical Xilinx ISE tools and a simple Makefile based automated development flow. The same structure works on both the Windows[1] and Linux operating systems. The template is in the `/opt/ufpapps/`*`vhdl_template`*`/src` directory. This design template offers the following features:

- Complete definition of all the FPGA external interfaces and pins
- Complete framework of makefiles for compiling, simulating, and building the design
- Separation of Cray HDL code and structures from user files
- Separation of Cray cores and simulation models from user files
- Straightforward integration into new Cray Inc. software releases

This directory structure is simple, flexible, and easy to maintain; see Figure 16, page 46. However, depending on the tool set that you choose for development, you may wish to modify the structure as required. For example, if a third-party synthesis tool is used, a separate `syn` directory may be useful to contain vendor-specific setup and control files.

1 To use the `makefile` environment on a Windows system, you can install the free Cygwin software package (see `www.cygwin.com`). It includes the GNU `make` package.

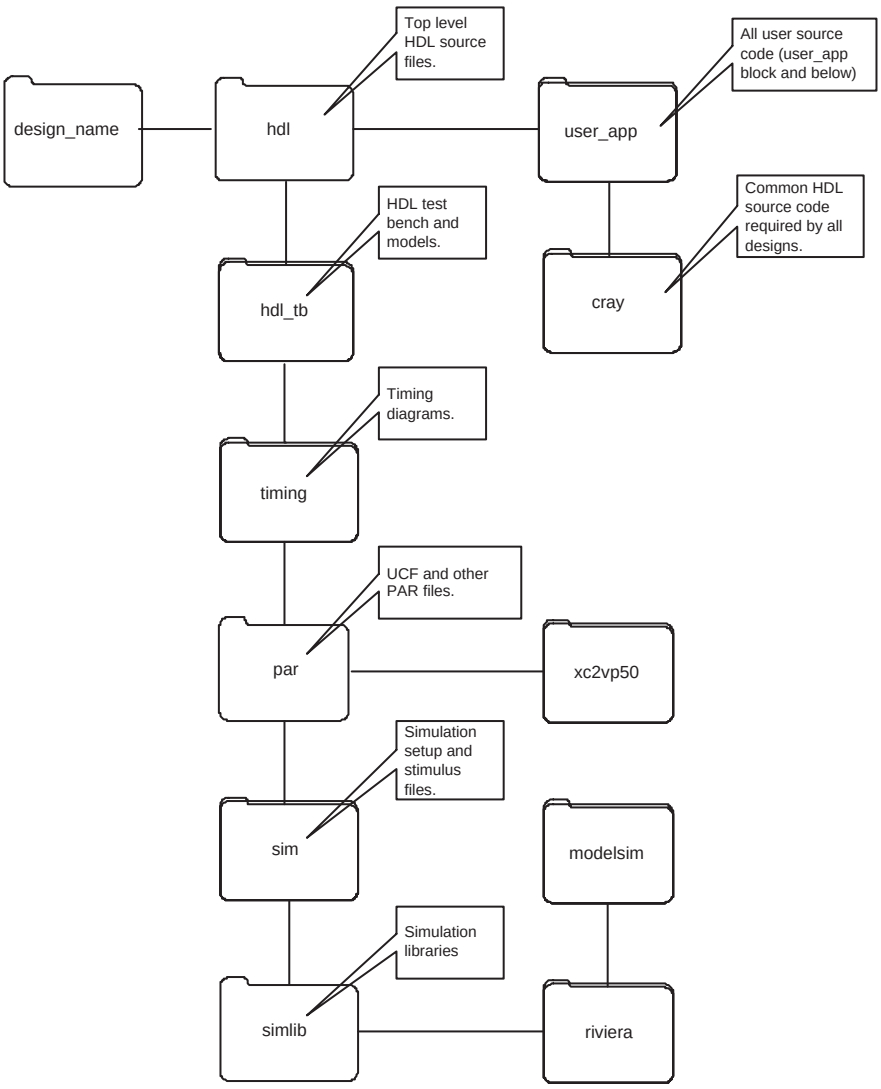

Figure 16. Design template directory structure

Table 7, page 47 describes the template design directories.

Table 7. Template design directory descriptions

Directory	Subdirectory	Description
hdl		Contains the top.vhd file which fully defines interfaces to the user application and Cray cores.
	user_app	Contains the source code for the FPGA user logic design.
	cray	Contains Cray VHDL code that is required in all designs.
hdl_tb		Contains a VHDL test bench for the design, source models for the QDR II SRAMs, and a simple behavioral model for the RapidArray processor
par	xc2vp50	Multiple directories that contain the synthesis and implementation files for the different variants of the AAP module
sim		Contains Modelsim scripts that are useful in the simulation of the design
	tc_01	Multiple directories that contain test case input and output files; each directory contains a separate test case that you can run individually or as part of a series for regression testing
	tc_02	
	…	
timing		Contains diagrams that illustrate the behavior of some design blocks; You can display these diagrams with a free viewer (see www.timingdesigner.com).
simlib		
	modelsim	Contains simulation libraries for the modelsim simulator
	riviera	Contains simulation libraries for the riviera simulator

7.3 Working with the Design Template

This section details how to use the design template in each step of the design process. You can work from either the command line or the GUI environment that the third-party tools provide. This process requires software packages that are not part of the Cray XD1 software distribution.

7.3.1 Tools

The design template works with specific tools. It can be used with other tools, but some modification will likely be necessary. Table 8, page 48 lists tools that you can use with the template with little or no modification.

Table 8. Recommended software packages

Type	Software package	In Cray XD1 software distribution?
Shell	Bash	Yes
Editor	Emacs, vi	Yes
Build management	Make	Yes
HDL Compilation	Aldec Riviera, Mentor Modelsim, Xilinx ISE	No
HDL Simulation	Aldec Riviera, Mentor Modelsim	No
Synthesis	Xilinx ISE	No
Place and Route	Xilinx ISE	No

7.3.2 Required Customizations

Table 9, page 49 describes the files in the design template that likely require customization.

Table 9. Design template customizations

File	Customization
`makefile_vars`	Update `makefile` variables to refer to your tools and paths.
`hdl/user_app/`*XXX*	Add user logic files to this directory.
`hdl/user_app/Makefile`	Update the `SOURCE` variable to contain all user logic HDL source files.
`par/`*arch*`/top.npl`	Add the appropriate hdl files (update by hand or use the GUI).
`par/`*arch*`/top.prj`	Add the appropriate hdl files (update by hand or use the GUI).
`par/`*arch*`/top.ucf`	Adjust the timing specifications to the desired operational frequency.
`sim/tc_`*XX*`/fabric.in` `sim/tc_`*XX*`/test.do`	Create a directory for each test case and update the `fabric.in` and `test.do` files in each directory as required.

7.3.3 Command Line Execution and Makefile Targets

The template includes a structure of makefiles that enable you to initiate all steps of the design process from the command line. These makefiles refer to the `makefile_vars` file at the top level of the design directory structure. This file centralizes the setting of some variables that all the makefiles use. Table 9, page 49 indicates that you need to customize the values of these variables for your configuration. Comments in the file describe the variables. Typically, you need to customize at least the following variables: `SIMULATOR`, `TOOL_ROOT`, and `UFPAPPS`.

After you customize the design template, you can use the make command with the targets described in Table 10, page 50.

Table 10. Makefile targets

Target	Description
`sim_setup`	Simulation setup. Creates the simulation libraries and directories that are prerequisites for simulation. Make this target before the first time simulation is run and every time that you add new HDL files to or remove them from the design.
`sim`	Batch mode simulation. Compiles required files and starts simulation in batch mode, which runs all test cases and reports error and warning assertions in the output. Test cases must be in the `sim` directory in subdirectories labeled `tc_01`, `tc_02`, and so on.
`sim test=DIR`	Interactive mode simulation. Compiles the required HDL files and starts the simulator GUI with the test case files as stimulus. Test cases must be in the `sim` directory in directories labeled `tc_01`, `tc_02`, and so on.
`xc2vp50`	Produce an FPGA binary. Synthesizes the HDL code, runs the logic mapper, and then the logic place and route tools. It is possible to compile logic for different FPGAs. This target is explicitly for the Xilinx XC2VP50 FPGA. For example `par/xc2vp50` is the directory that stores information for constructing the FPGA binary for the Xilinx Virtex 2 Pro 50.
`clean`	Clean up temporary files. Removes temporary files from the design directories.
`clean_all`	Clean up temporary and intermediate files. Removes all temporary and intermediate synthesis files from the design directories.

To initiate one of these steps:

1. Move to the directory that holds the HDL design files (for example, `design_name` in Figure 14, page 42).
2. Run the `make` command:

   > **make** *target*

   where *target* is one of the entries in Table 10, page 50.

   Example 1: Using makefile targets

   This example sets up the simulation files and then starts the simulator GUI for interactive verification.

   ```
   > cd /path/to/mydesign/src> make sim_setup...
   > make sim test=tc_01
   ```

7.3.4 Using the Xilinx ISE GUI

You can run the commands that produce an FPGA binary from either the Linux shell or from the Xilinx ISE GUI. The GUI requires a project file to run. The project file is in the `par` subdirectory of the target model FPGA (for example, `par50`). Its name is `top.npl`. Describing how to use the third-party tools is beyond the scope of this document. For information on using the Xilinx ISE GUI, refer to the Xilinx documentation that is provided with the tool or on the Xilinx website (`http://www.xilinx.com`).

7.3.5 Simulating the Design

You can simulate a design by instantiating it and connecting it to simulation models that provide stimulus. The design template includes a test bench in the `hdl_tb` directory; its name is `top_tb.vhd`. Figure 17, page 52 shows the structure of the test bench.

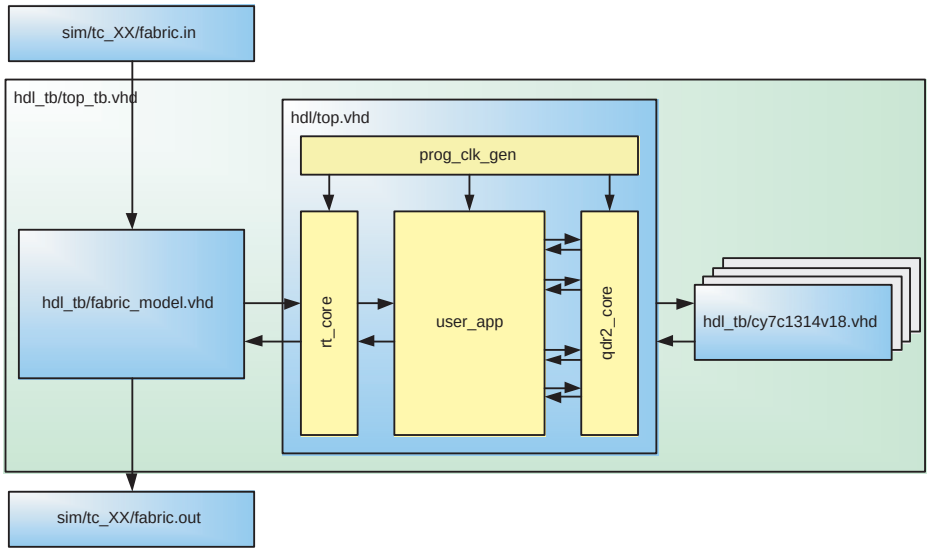

Figure 17. HDL test bench organization

Stimulus for the fabric model changes depending on which test case you run. The stimulus files for a test case and the output from the last time the test case ran are in the `tc_` *XX* (for example, `tc_01`) subdirectory of the `sim` directory. For the fabric model, these input and output files are called `fabric.in` and `fabric.out` respectively.

The `test.do` file in the `tc_` *XX* subdirectory contains a list of simulator commands to execute when the test case runs. The simulator run time is an important parameter to verify if you run the test case in batch mode. For further details on using the simulation models, see Chapter 8, page 63.

7.4 Interfacing User Logic to Other Cray XD1 Resources

The FPGA user logic interfaces to the Cray XD1 system through the RT core and the QDR II SRAM core. The application design determines which cores or portions of cores that you need.

Often, the purpose of the FPGA AAP is to accelerate a software algorithm. The challenge is to write part of the algorithm in HDL code and then interface the user logic to the system so that a program on an SMP can use it. The interface between the application and the FPGA user logic consists of data flows and the control structures that manage them.

When you identify a suitable application or algorithm, examine the data flows and resources that it requires. There are several ways to move data between the SMP and the FPGA user logic. To decide where to store data and how to move it between the SMP and the FPGA logic, keep the suggestions from Chapter 6, page 33 in mind. Consider the following questions:

- How will the SMP acquire the data (for example, from a disk or network)?
- How fast must the application process data?
- What is the structure of the data?
- How will the software move the data to the FPGA and in what form?
- Where will the FPGA store intermediate and final results?
- How will the FPGA return results to the SMP?
- How will the SMP acquire the data (for example, from a disk or network)?
- How fast must the application process data?
- What is the structure of the data?
- How will the software move the data to the FPGA and in what form?
- Where will the FPGA store intermediate and final results?
- How will the FPGA return results to the SMP?

For a description of the methods of moving data between the SMP software application and the FPGA user logic, see Section 7.5, page 54.

After you identify the data flow and data structures, consider the control mechanisms and structures that you need.

Control mechanisms are dependent on the data flow but typically include configuration registers and ring buffers. For example, you can use configuration registers to tell the FPGA logic the locations of data structures in the SMP memory and when to start processing.

When you know the data and control requirements, you can create a memory map. This map defines where in the FPGA address space to locate the data and control resources; applications on the SMP also use this map. The mapping of resources is flexible—you can tailor it to each design.

At the minimum, user logic must use the Fabric Request Interface of the RT core to respond to the SMP software application. The user logic that connects to the interface compares the request addresses to the memory map and takes appropriate action. Some examples of requests are accessing an internal register or block ram and translating and forwarding a request to an interface of the QDR core. The hello reference design demonstrates some basic examples of how to handle these accesses.

Accesses to certain resources may require arbitration. This occurs if two portions of user logic can access the resource or if both the application on the SMP and part of the user logic can access the resource. The user logic is responsible for arbitration whenever it is required.

7.4.1 Disabling Unused Core Interfaces

If the user logic does not require the use of some of the interfaces on a core or an entire core that is instantiated in `top.vhd`, force the unused signals to an unchanging inactive state from the `user_app/user_app.vhd` file. This causes the HDL compiler to optimize out the unnecessary logic. The `hello` reference design includes some examples.

7.5 Interaction with the SMP Software Application

This section explains the relationship between API calls from a software application on the SMP and the RT core bus transactions that are delivered to the user logic in the FPGA. For further details of the C API function calls and the RT core bus transactions, see *Cray XD1 Programming* (S–2433) and *Design of Cray XD1 RapidArray Transport Core* (S–6411), respectively.

In the Cray XD1 architecture, the FPGA AAP is an RT fabric device. All accesses to or from the FPGA happen across the RT fabric.

Software applications on the SMP access the FPGA by calling the API, which initiates appropriate transactions on the RT fabric. These transactions travel through the RT fabric and are delivered to the RT core in the AAP. It is the responsibility of the user logic in the AAP to process the transaction and respond appropriately.

User logic in the FPGA can also access the SMP memory through the RT core. The user logic in the FPGA sends a bus transaction to the RT core, which forwards it through the RT fabric to hardware on the SMP, where it becomes a read or write transaction to the SMP DRAM.

7.5.1 Memory Map

Because the RAP effectively connects the FPGA to the local SMP's HyperTransport link, the FPGA is accessible via a region of the HyperTransport I/O address space. Specifically, the FPGA occupies a 128 MB address region of the link. Any HyperTransport read and write requests issued by the SMP to this region are directed to the RT interface of the FPGA which passes them on to the user logic. From a logical perspective, the FPGA appears similar to a PCI device in a legacy system, but, from a performance perspective, it appears as if the FPGA connects directly to the SMP.

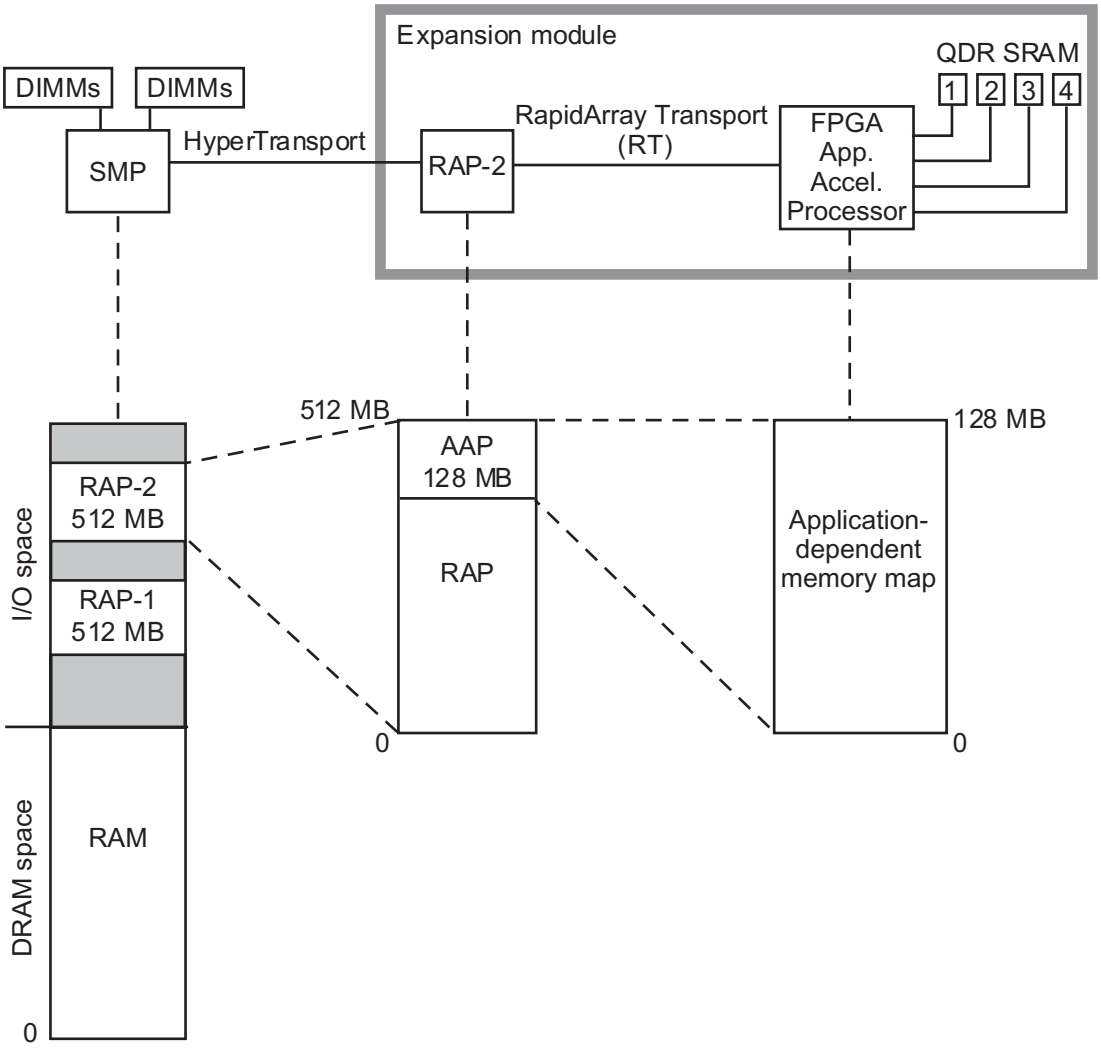

Figure 18. Physical components and related address spaces

7.5.2 Software API Commands

Table 11, page 57 lists the C API functions for the FPGA AAP. For more information on the API, see *Cray XD1 Programming* (S–2433).

Table 11. C API functions

API function name	Description
`fpga_open`	Opens an FPGA device
`fpga_load`	Loads a converted binary logic file into an FPGA
`fpga_is_loaded`	Queries the programming state of an FPGA
`fpga_reset`	Places the FPGA user logic into reset
`fpga_start`	Releases the FPGA user logic from reset.
`fpga_memmap`	Maps a region of the FPGA address space into the application address space
`fpga_mem_sync`	Forces completion of all outstanding transactions to mapped FPGA memory
`fpga_register_ftrmem`	Registers a region of application memory for direct access by FPGA
`fpga_dereg_ftrmem`	Deregisters an FPGA transfer region
`fpga_set_ftrmem`	Sets up a region of memory in application process for direct access by FPGA **Note:** This function is deprecated. Cray recommends that you use the `fpga_register_ftrmem` function instead.
`fpga_rd_appif_val`	Reads a value from the FPGA address space and guarantees access order
`fpga_wrt_appif_val`	Writes a value to the FPGA address space and guarantees access order
`fpga_status`	Gets the status of an FPGA device
`fpga_unload`	Clears the programming of an FPGA
`fpga_close`	Closes an FPGA device

7.5.3 SMP-initiated RT Fabric Requests

The software application can initiate transactions to the FPGA in two ways. It can either map the FPGA into the address space of the SMP and then make ordinary memory references or it can call specific read and write functions from the API.

Note: The important difference between these two access methods is that the I/O mapped accesses allow write combining while the read and write functions calls do not.

7.5.3.1 I/O Mapped Accesses

I/O mapped accesses take advantage of a feature called "write combining." Write combining greatly improves the performance of write accesses from the SMP to the FPGA by combining multiple write accesses into a single HyperTransport packet. The processor hardware does this by identifying writes to sequential address regions and packing them into a single burst write transaction.

One important side effect of write combining is that writes may not occur in the same order in which they were issued. For example, if writes are made to addresses 1, 2, 4, 3 in that order, the write combining feature re-orders the transactions into a sequential burst of 1, 2, 3, 4.

This type of access is often useful when the program accesses general-purpose memory because the order of the writes is often not important. In the cases where order is important, the application program can either call the `fpga_mem_sync`(3) function which enforces order by inserting a memory fence or call the `fpga_rd_appif_val`(3) and `fpga_wrt_appif_val`(3) functions.

7.5.3.1.1 Software API Details

Memory mapping the FPGA and then reading and writing to a location in the mapped memory requires the software application to make the function calls shown in Example 2, page 58. For simplicity, the code and function calls listed below are not complete. For complete information on the functions and their use, see *Cray XD1 Programming* (S–2433).

Example 2: I/O mapped writes to the AAP

```
u_64 fpga_mem, i;

my_fpga = fpga_open(args);
...
if (!fpga_is_loaded(args)) {
 rtn = fpga_load(args);
```

```
}
...
fpga_mem = fpga_memmap(args);
...
for (i = 7; i >= 0; i--;) {
 fpga_mem[i] = i;
}
...
```

In Example 2, page 58, the program maps the FPGA and then writes a sequence of 64 bit values to locations that are offset from the base of the I/O mapped region. The program writes the value 7 to offset 7, the value 6 to offset 6, and so on down to offset 0.

7.5.3.1.2 Hardware API Details

The accesses shown in Example 2, page 58 are memory mapped. Therefore, the processor reorders and combines them into a burst access to the FPGA. The user logic on the FPGA receives a write request from the fabric request interface of the RT core. The request is a single burst write of eight quadwords to address 0 with data values 0, 1, 2, 3, 4, 5, 6, and 7. This effectively addresses 0x0, 0x8, 0x10, 0x18, 0x20, 0x28, 0x30, 0x38 because quadwords are addressed. For more information on the RT core interface, refer to *Design of Cray XD1 RapidArray Transport Core* (S–6411).

7.5.3.2 API Function Accesses

The API functions `fpga_wrt_appif_val`(3) and `fpga_rd_appif_val`(3) do not allow write combining and enforce the order of reads and writes. Use these functions when you access control and status information because they ensure that all transactions complete in order. These functions are not as efficient as memory-mapped accesses because they enforce transaction order. Therefore, they are not suitable for large data transfers.

7.5.3.2.1 Software API Details

Example 3, page 60 shows how an application can use the `fpga_wrt_appif_val`(3) function. For simplicity, the code and function calls in the example are not complete. For complete information on the functions and their use, see *Cray XD1 Programming* (S–2433).

Example 3: Accessing the AAP with `fpga_wrt_appif_val`

```
u_64 fpga_mem, i;

my_fpga = fpga_open(args);
...
if (!fpga_is_loaded(args)) {
 rtn = fpga_load(args);
}
...
for (i = 7; i >= 0; i--;) {
 fpga_wrt_appif_val(my_fpga, i, i, TYPE_VALUE, &err);
}
```

In Example 3, page 60, the program loads the FPGA and then writes a sequence of 64 bit values to the FPGA address space at quadword offsets 7 down to 0 (addresses 0x38 down to 0x0).

7.5.3.2.2 Hardware API Details

The program in Example 3, page 60 uses the `fpga_wrt_apapif_val` function; therefore, the order of the write transactions is preserved. The user logic on the FPGA receives eight single quadword write requests from the RT core fabric request interface in order. The first is a quadword write to offset 0x38 with a value of 7. For more information on the RT core interface, refer to *Design of Cray XD1 RapidArray Transport Core* (S–6411).

7.5.4 FPGA-initiated RT Fabric Requests

The FPGA can access the local memory of the SMP. To do so, the FPGA user logic requires initialization information from the application program on the SMP. At a minimum, this initialization includes a pointer to the shared memory buffer in the SMP DRAM. The software must write this pointer to the FPGA with the `fpga_wrt_appif_val`(3) function. The *type* parameter of the function must have the value 1, which translates the address to the type required by the FPGA.

User logic in the FPGA can use the RT core User Request Interface to generate read and write requests to the SMP. Writes are posted and can be burst transactions of up to eight quadwords. Unlike the SMP, the AAP user logic has full control over the ordering and all other aspects of the write transactions through the RT core. Also unlike the SMP, the AAP can issue burst read requests of up to 64 bytes and issue multiple outstanding read requests. By

issuing multiple burst read requests, the FPGA can achieve full utilization of the HyperTransport link bandwidth when it reads from the SMP memory.

Example 4: Initializing the AAP to access the SMP memory

```
u_64 fpga_mem, i;

my_fpga = fpga_open(args);
...
if (!fpga_is_loaded(args)) {
 rtn = fpga_load(args);
}
...
ftr_mem = fpga_set_ftrmem(args);
...
fpga_wrt_appif_val(my_fpga, SMP_MEM_PTR_REG, ftr_mem, 1, &err);
```

Example 4, page 61 shows how a program creates an FPGA transfer region and then writes a pointer to that region to a register location, SMP_MEM_PTR_REG, in the FPGA user logic. For more information, see *Design of Cray XD1 RapidArray Transport Core* (S–6411).

Simulation and Debugging [8]

This chapter describes some of the methods available to assist you in debugging user logic on the FPGA application acceleration processor. Cray provides both simulation models and a hardware interface to the Xilinx JTAG port for the purpose of debugging and verification.

8.1 Simulation Models

Simulation models for all the Cray XD1 cores as well as a simple behavioral model for the RT fabric are included in the Cray XD1 software distribution. These models are in the design template simulation test bench.

8.1.1 Cray XD1 Core Simulation Models

Simulation models are present for each of the Cray cores. The models come in several formats:

- An encrypted Riviera model
- A VHDL netlist file

Encrypted models should perform faster during simulation, but you can use them only with the specified simulation software package. You can use VHDL netlists with any simulator.

8.1.2 RT Fabric Behavioral Model

The behavioral model for the RT fabric is a simple VHDL model that you can use to simulate read and write requests to and from the SMP SDRAM and processors. Typically, designers modify the fabric model inputs or the fabric model itself to properly verify the user logic. Examples of how to use the RT fabric model are in each reference design.

The fabric model inputs are read from the `fabric.in` file. Requests from the fabric model appear at the fabric request interface of the RT core where the user logic must handle them. Read requests can be compared against expected values and assertions will be raised when the expected and actual values differ.

The `fabric.in` file may contain several commands. Table 12, page 64 lists the fabric commands; the `fabric.in` file also documents these.

Table 12. Commands supported in the `fabric.in` file

Command name	Command format
Initialize Link	`I`
Print	`P` *text_to_print*
Delay	`D` *delay_value*
Read	`R` *address expected_data byte_mask byte_request size*
Write	`W` *address write_data byte_mask byte_request size*
Burst	`B` *data byte_mask*

Table 13, page 64 describes the `fabric.in` command arguments.

Table 13. Arguments of the `fabric.in` commands

Argument	Description
text_to_print	A text value. The text to print to the console.
delay_value	An integer value greater than 1. The number of user clock cycles to delay.
address	A 40-bit value. The hex address to access.
expected_data	A 64-bit value. The data read is compared to this value and an assertion is raised if the values differ.
byte_mask	An 8-bit value. Controls which bytes of the data to access. The least significant (furthest right) bit, bit 0, controls the least significant byte (furthest right) byte 0. A value of logic 1 enables the byte.
byte_request	A single-bit value. When the byte request is a logic 1, the byte mask is used to determine which bytes to access. When byte request is logic 0, a quadword request is made.
size	A single-byte value. The size of the request in doublewords. A value of 0x0 indicates a single doubleword transfer, while a value of 0xF indicates a 16 doubleword transfer.

Some additional conventions that these input files use are as follows:

- All numeric values are in hexadecimal notation.
- The # character introduces a comment.

Example 5, page 65 shows a `fabric.in` file. The example performs the following actions:

- Delay for 100 user clock cycles.
- Print a string that describes the read which follows.
- Burst read that starts at address 0x0004000000 and is 16 doublewords long. A read command must begin with a read line and then follow with enough burst lines to equal the size of the access. The first read line and each subsequent burst line includes the expected value from each subsequent location.
- Print a string that describes the write which follows.
- Burst write that starts at address 0x0004000018 and is 10 doublewords long. A write command must begin with a write line and then follow with enough burst lines to equal the size of the access. Each line of write command includes a data value and a byte mask, which form the value that is written to that address.
- Print a string that describes the read that verifies the write.
- Burst read that starts at address 0x0004000000 and is 16 doublewords long.

Example 5: Format of the `fabric.in` file

```
##########
# Insert a delay to allow the DLLs some time to lock
D 100
##########
# Test access to the internal register space. The internal
# registers are mapped by the RT Client into the register space
# (i.e. $00_0400_0000 to $00_07FF_FFFF, or the top 64 Mbytes);
####
P Fabric Model: Testing a burst read of the registers
####
R 0004000000 0000000300360100 FF 0 F # Read base, revision, etc
B 0000000000000000 FF # Read the App. Config
B 0000000000000000 FF # Read the App. Latch
B AAAAAAAAAAAAAAAA FF # Read the user register 1
```

```
B BBBBBBBBBBBBBBBB FF # Read the user register 2
B CCCCCCCCCCCCCCCC FF # Read the user register 3
B DDDDDDDDDDDDDDDD FF # Read the user register 4
B EEEEEEEEEEEEEEEE FF # Read the user register 5
##########
P Fabric Model: Testing a burst write to the registers
####
W 0004000018 0123456789ABCDEF FF 0 9 # Write user register 1
B 55AA55AA55AA55AA FF # Write user register 2
B FEDCBA9876543210 FF # Write the user register 3
B FFFFFFFFFFFFFFFF FF # Write the user register 4
B AA55AA55AA55AA55 FF # Write the user register 5
##########
P Fabric Model: Reading back the registers
####
R 0004000000 0000000300360100 FF 0 F # Read base,revision,etc
B 0000000000000000 FF # Read the App. Config
B 0000000000000000 FF # Read the App. Latch
B 0123456789ABCDEF FF # Read the user register 1
B 55AA55AA55AA55AA FF # Read the user register 2
B FEDCBA9876543210 FF # Read the user register 3
B FFFFFFFFFFFFFFFF FF # Read the user register 4
B AA55AA55AA55AA55 FF # Read the user register 5
```

8.1.2.1 FPGA Transfer Region

The FPGA transfer region (FTR) is a portion of the SMP processor DRAM that is shared with the FPGA—see Example 4, page 61. The fabric model simulates this region using the `ram` signal, which is of type `qw_array` for quadword array. If you need initial values in the FTR, modify the fabric model `ram` signal accordingly.

8.2 Using the JTAG Interface Card

In addition to design verification through simulation, it can be necessary to verify and debug the design on the target hardware. Many types of problem are quite difficult and time consuming to anticipate, recreate, and test in a simulation environment. To facilitate target debugging, the Cray XD1 system provides access to the JTAG port of the FPGA.

8.2.1 The JTAG Interface Card

Access to a JTAG port occurs through an optional JTAG interface card (see Figure 19, page 67), which plugs into one of the high-speed I/O slots in the back of the Cray XD1 chassis.

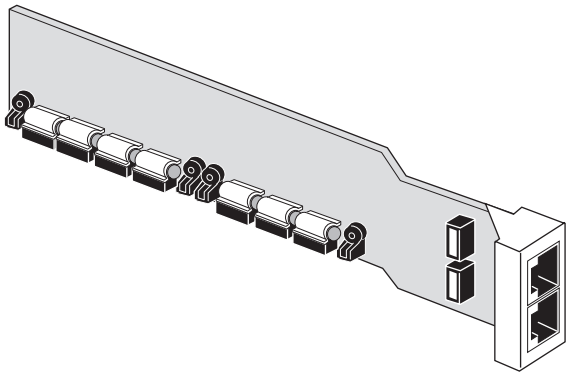

Figure 19. JTAG interface card

Note: The Cray XD1 product warranty requires that only Cray service personnel, Cray-authorized service providers, or Cray-trained customers perform hardware maintenance. See the `http://crinform.cray.com/xd/` website for information on Cray XD1 training programs and service support. If you install a JTAG interface card in the field, read and follow the field replacement procedure that comes with the card.

8.2.2 Mapping JTAG Interface Ports to FPGAs

The three I/O slots on the back of the Cray XD1 chassis are numbered 1 through 3 from left to right as seen from the back of the chassis. Each JTAG interface card has two ports that are numbered 1 and 2 from top to bottom. Figure 20, page 68 shows the configuration. Note that this figure shows three JTAG interface cards installed; there may be fewer in your system. Typically, a PCI-X expansion card occupies I/O slot 1.

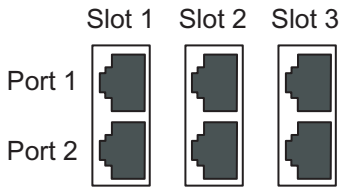

Figure 20. Ports on the JTAG interface card

Each port can connect to one of the six possible FPGAs in a chassis. Table 14, page 68 lists the default mapping of JTAG ports to FPGAs.

Table 14. JTAG interface card default connections

	Slot 1	**Slot 2**	**Slot 3**
Port 1	Node 1	Node 3	Node 5
Port 2	Node 2	Node 4	Node 6

If the default mapping is appropriate, you can proceed to connect your workstation to the relevant port with no further configuration task. Otherwise, you can use a Cray XD1 command to manage the connections of JTAG interface ports to FPGAs. You can view the current mapping of ports to FPGAs in a chassis, map any port to any FPGA in the chassis, or restore the default mapping of ports to FPGAs in a chassis.

8.2.2.1 Viewing JTAG Interface Port Connections

Use the following command to view the current connections of JTAG interface ports to FPGAs:

```
> xd1_jtag_route --chassis=chassis-addr --show
```

where *chassis-addr* is the address of the chassis in one of the following forms:

- IP address: *aaa* **.***bbb***.** *ccc***.***ddd*

  This address appears on the first line of the LCD on the front of the chassis.

- Host name: **chassis***chassis-id*

  where *chassis-id* is the hardware ID (serial number) of the chassis. An example of the chassis host name is `chassis371`.

The command lists the current connections and the state (enabled or disabled) and status of each.

8.2.2.2 Connecting a JTAG Interface Port to an FPGA

Use the following command to map a JTAG interface port to a particular FPGA in a particular chassis:

```
> xd1_jtag_route --chassis=chassis-addr --slot=slot-num
  --port=port-num --aap=node-num
```

where *chassis-addr* is as described in Section 8.2.2.1, page 68, *slot-num* is the number of the slot that the JTAG interface card occupies, *port–num* is the number of the relevant port on the card, and *node-num* is the ordinal of the target node (and hence the FPGA) in the chassis.

Example 6: Connecting a JTAG interface port to an FPGA

To connect the upper port on a JTAG interface card in I/O slot 2 of chassis 371 to the FPGA on node 5 of that chassis:

```
> xd1_jtag_route --chassis=chassis371 --slot=2 --port=1 --aap=5
```

8.2.2.3 Restoring the Default JTAG Interface Port Connections

Use the following command to restore the default mapping of JTAG interface ports to FPGAs:

```
> xd1_jtag_route --chassis=chassis-addr --default
```

where *chassis-addr* is as described in Section 8.2.2.1, page 68.

The command resets the mapping of all JTAG interface ports to the default shown in Table 14, page 68.

8.2.3 Connecting a Workstation to a JTAG Interface Port

The JTAG port enables you to use tools such as the ChipScope Pro debugger from Xilinx with the FPGA. ChipScope provides logic analyzer-like functionality—it enables you to monitor and capture logic events as they occur in the FPGA. For more information on ChipScope, refer to the Xilinx website at `http://www.xilinx.com`.

Currently, the ChipScope Analyzer software runs only on the Windows operating system of a PC. It connects to the target device through the parallel port of the PC. Cray provides an RJ-45 to DB25 adapter that enables you to connect a CAT 5 cable between a port on the JTAG interface card and the PC parallel port. Figure 21, page 70 illustrates the adapter.

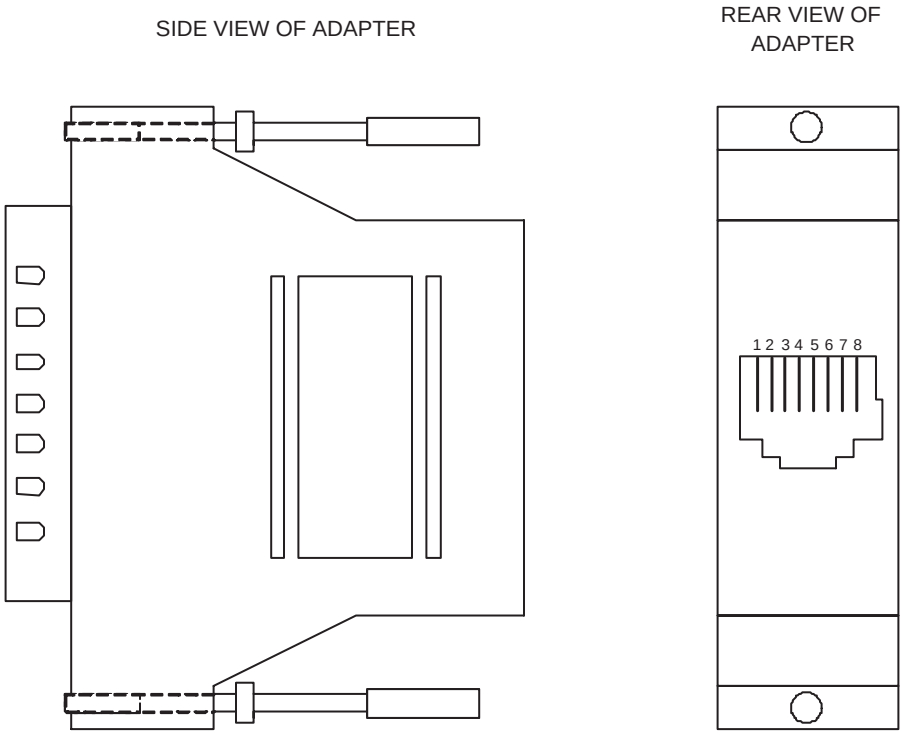

Figure 21. JTAG I/O adapter

The JTAG adapter plugs into the parallel port of a PC, and a standard CAT5 (straight-through) networking cable connects the adapter to the JTAG interface card in a Cray XD1 system; see Figure 22, page 71.

Caution: The CAT 5 connector and cable are identical to Ethernet equipment. Do not accidentally plug networking equipment into the JTAG interface card. Damage to the networking equipment and/or the JTAG interface card may occur.

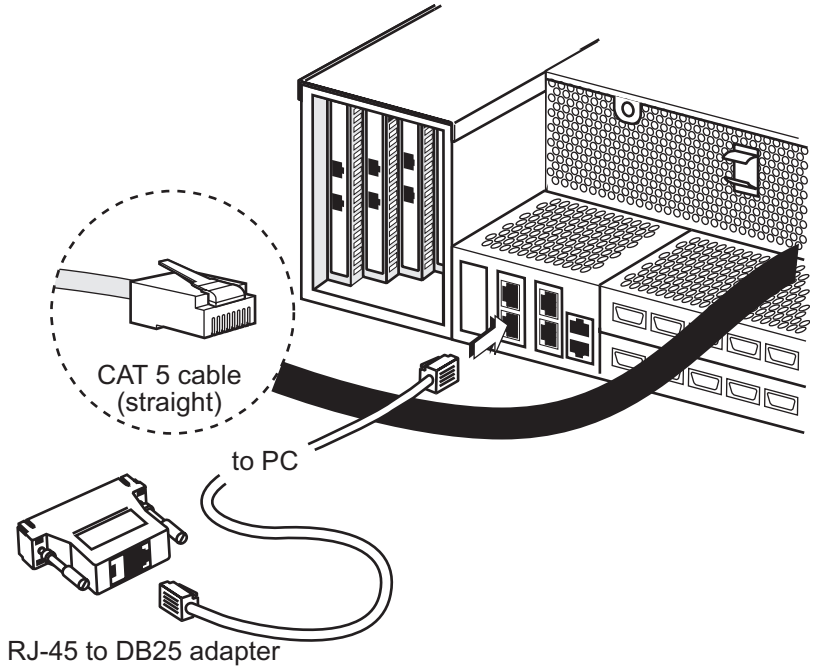

Figure 22. Accessing the JTAG interface card

Troubleshooting [9]

This chapter describes some issues that may occur with the FPGA application acceleration processor and how to resolve them.

9.1 Node Hangs After Accessing `/proc/ufp/regs`

9.1.1 Cause

The FPGA AAP did not respond to an SMP processor read request.

9.1.2 Discussion

If an FPGA AAP is programmed but the user logic is in a reset state (via the `fcu --reset` command or the `fpga_reset`(3) function call), the FPGA APP cannot respond to SMP processor read requests. If a request is made, the SMP processor experiences a bus timeout and reboot.

Use the reset functionality with care. If possible, restrict its use to debugging. If you erase the device (via the `fcu --unload` command or the `fpga_unload`(3) function call), the bus transaction is handled correctly.

This behavior may change in a future release.

9.2 File `/dev/ufp0` Does Not Exist (Interaction with the FPGA AAP Fails)

9.2.1 Cause

The FPGA AAP kernel module is not properly installed.

9.2.2 Discussion

The Opteron processors access the FPGAs through a standard Linux device driver. The Cray XD1 Active Manager software automatically detects the expansion module on a compute blade and loads the driver. If the Active Manager software is not running or fails to install the driver, you can load the module manually in the same way as any other Linux device driver. The

following procedure describes how to configure the kernel to load the driver automatically when it is required.

Note: This procedure is required only if Active Manager is not in use. If Active Manager is running, these entries will already exist.

Procedure 1: To load the FPGA driver

1. Log in to the relevant node of the Cray XD1 system as `root`.
2. Create the `ufp` device:

   ```
   # mknod /dev/ufp0 c 62 0
   ```

3. Add the following three lines to the `/etc/modules.conf` file:

   ```
   alias char-major-62 ufp
   alias ufp0 ufp
   options ufp cm_part_num=module-number
   ```

   where *module-number* is the Cray part number of the target module in quotation marks (for example, `"90-0003-05"`). Refer to Table 3, page 13 for a list of available modules.

After the operating system loads driver, the FPGA special file `/dev/ufp0` appears. The FPGA device driver will load when it is required. Generally this occurs when a user application or the `fcu`(1) utility loads the FPGA. You can force the driver to load by running the `fcu -r` . After the driver is loaded, the virtual files `/proc/ufp/fpga` and `/proc/ufp/regs` appear on the SMP.

Glossary

AAP

Application acceleration processor. See *FPGA application acceleration processor*.

Active Manager

The software that monitors and manages all aspects of the Cray XD1 system. Its user interfaces provide administrators and end users with a single point of control for the system.

chassis ID

The permanent numeric identifier of a chassis, unique to each Cray XD1 chassis. A chassis ID has a maximum of six decimal digits.

compute blade

One of six circuit boards in a Cray XD1 chassis; contains Opteron processors configured as an SMP, DIMMs, and a RapidArray processor. A compute blade may also have an expansion module.

Cray XD1 system

A stand-alone Cray XD1 chassis or multiple chassis that communicate over both the supervisory network and the RapidArray interconnect.

crossbar switch

A communications switch that provides direct connection between any pair of ports.

DDR SDRAM

Double data rate synchronous dynamic random-access memory.

doubleword

Four bytes (32 bits).

expansion module

Optional Cray XD1 hardware that connects to each compute blade; if they are

present, a chassis has six expansion modules. The expansion modules provide a node with a second RapidArray processor, two additional Rapid Array links, and an optional application acceleration processor.

fabric

The collection of fabric components that interconnect in the same switching plane. A Cray XD1 system has one or two independently wired, parallel RapidArray fabrics: the main fabric and the optional expansion fabric. These fabrics are also known as fabric X and fabric Y, respectively.

fabric expansion card

Optional hardware in a Cray XD1 chassis that adds a second RapidArray fabric to the system: provides a second internal 24-port RapidArray switch, 12 additional internal links, and 12 additional external ports for chassis interconnection. The fabric expansion card connects to the main board.

field-programmable gate array

An integrated circuit that consists of arrays of AND and OR gates (typically thousands) that can be programmed to perform complex functions. The Cray XD1 system has optional FPGAs available for use as application acceleration processors.

FPGA

See *field-programmable gate array*.

FPGA application acceleration processor

An FPGA that users can program to accelerate computationally intensive and repetitive algorithms; acts as a co-processor to the Opteron processor. This is an optional component on the expansion module. See also *JTAG interface card*.

interconnect

See *RapidArray interconnect*.

JTAG interface card

Optional hardware that tests the application acceleration processor's integrated circuits. This card connects to one of the high-speed I/O slots on the main board of a Cray XD1 chassis.

link

See *RapidArray link*.

management processor

The processor on the main board of a Cray XD1 chassis that runs the Active Manager hardware supervisory subsystem.

node

An instance of the Linux operating system and the hardware components that it controls. The hardware components in a Cray XD1 node include an SMP and its associated memory, one or two RapidArray processors (depending on configuration) and, optionally, an FPGA application acceleration processor.

PCI-X expansion card

Hardware in the Cray XD1 chassis that provides four PCI-X slots for Gigabit Ethernet and Fibre Channel cards. This card also provides connectors for three disk blades. It connects to one of the three high-speed I/O slots on the main board.

quadword

Eight bytes (64 bits).

RAP

RapidArray processor.

RapidArray interconnect

The high-speed network that interconnects the nodes in a Cray XD1 chassis, and connects all nodes in a Cray XD1 system via cables and optional external RapidArray switches. The RapidArray interconnect consists of a main and an optional expansion fabric, each with its own set of fabric components. The configuration of the RapidArray interconnect in a multichassis system is called the physical topology.

RapidArray link

The physical communication path between two RapidArray ports. Each link can carry two gigabytes per second.

RapidArray processor

The special-purpose processor on a Cray XD1 compute blade; responsible for most communication functions within the system. The RapidArray processor interfaces an Opteron processor to the RapidArray fabric.

RapidArray switch

A full-crossbar nonblocking switch in the RapidArray fabric. The base configuration includes one 24-port RapidArray switch in each Cray XD1 chassis. The optional fabric expansion card adds a second RapidArray switch. Equivalent external RapidArray switches are available for implementing fat tree (switched) topologies.

RapidArray Transport core

An IP core for the FPGA application acceleration processor that provides the logic necessary for an FPGA design to interface (via the RapidArray fabric) to the rest of the Cray XD1 system.

SMP

Symmetric multiprocessor.

switch fabric

See *fabric*.

ufp

User FPGA processor. A combining form that occurs in file and directory names; for example, in `libufp.a`. It is a synonym for the FPGA application acceleration processor.